tech

COLLABORATORS			
	TITLE : tech		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 8, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

tech

1.1 3 How TCS Works

```
                3   How TCS Works


*************************************************************************
*I must stress clearly that everything that follows (here and in the rest*
*of the guide) derives directly from _practice_, as I don't know anything*
*of optics and the likes; that's why it's likely that some terms are used*
*improperly - I apologize for this. It could well also be that not even a*
*subtle shade of truth backs up my resoning: all I know is that it _does_*
*work as a matter of fact. I must apologize again for this lack of scien-*
*tific rigour.                                                          *
*************************************************************************


Here you can find all the technical details to perfectly understand the
inner workings of TCS and code your own set of routines to operate it
(in case you don't want to use the library).


3.1
                Basic Idea
                3.2
                Amiga Hardware Setup
                3.3
                RGBx Color Composition
                3.4
                Improving Picture Quality with ChqrMode
                3.5
                Creating Scrollable Screens
                3.6
                Cross Playfield Mode
                3.7
                Screen Buffering


 - for convenience, in all the following sub-sections, we assume to have
   an Amiga 320x256 LORES PAL display on which a TCS 160x256 screen in
```

HalfRes mode or a TCS 320x256 screen in FullRes mode is shown
- the sections above discuss rather simple concepts, but aren't easy to read at all: I couldn't do any better, many apologies
- all the methods suggested herein are implemented in the tcs.library
- although this part is quite vast and important, YOU DO NOT NEED TO DROWN YOURSELF IN IT to be able to use the tcs.library: understanding the basic concepts and terminology is fairly enough

1.2 3.1 Basic Idea

3.1 Basic Idea

The basic idea is to exploit the extremely small size of SHRES (35 ns) pixels. By opening a SHRES screen, in fact, pixels are so tiny that the human eye can't clearly distinguish the ones close to one another, so that it "mixes" their colors, thus perceiving a single pixel.
Suppose that a LORES pixel is represented by something like:

```
RRRR
RRRR
RRRR
RRRR
```

and a SHRES pixel by:

```
R
R
R
R
```

The eye can distinguish two LORES pixels attached to each other (provided their contrast is quite good):

```
RRRRGGGG
RRRRGGGG
RRRRGGGG
RRRRGGGG
```

But it finds quite difficult to "separate" two or more SHRES pixels:

```
RGRBGRBG
RGRBGRBG
RGRBGRBG
RGRBGRBG
```

This can be used to form different colors by attaching pixels with Red, Green and Blue tonalities:

```
RGB
RGB
RGB
RGB
```

these 3 SHRES pixels are perceived as a single 105 ns pixel whose color derives from the composition of the R, G and B components.

1.3 3.2 Amiga Hardware Setup

3.2 Amiga Hardware Setup

This section lists the fundamental Amiga hardware settings required to open a Tricky-Color display.

The Amiga chipset it so flexible that it allows to set up some really fancy displays. To exploit this we have to:

1. open a 1280x256, 4 [or 5, see 2.c] planes, SHRES display

2. a) reserve enough CHIP ram for 1 [HalfRes] or 2 [FullRes] 1280x256 bitplanes (named "VdoPln0" and "VdoPln1", "VdoPlns" in general)

   b) reserve some more CHIP ram for other 2 planes, both sized like VdoPln0, which act as color-component selectors, hence their names will be "SlcPln0" and "SlcPln1" ("SlcPlns" in general)

   c) [HalfRes] reserve, optionally, one more bitplane (warmly recommended) of the same size of VdoPln0 again, called "MskPln", whose purpose will be detailed in
   section 3.3.1.3
   d) [FullRes] reserve a buffer (in FAST mem, otherwise ↩ performance
   drops too much) that will be our 320x256 chunky screen (let's call it "ChnkScr")

3. set the BPLxPT in this way:

   BPL1PT = VdoPln0 address
   BPL2PT = VdoPln0 [HalfRes] or VdoPln1 [FullRes] address
   BPL3PT = SlcPln0 address
   BPL4PT = SlcPln1 address
   BPL5PT = MskPln address (only if required)

   (for
   Cross Playfield
   these settings have to be extended
   this way
)

4. a) HalfRes: set BPLCON1 to $10, so that playfield 2 (planes 2,4) is shifted by 1 LORES pixel with respect to playfield 1 (planes 1,3,5);
   this can be changed to $10 for even lines and $21 for odd ones, in order to achieve a "
   chequered
   " display to avoid
   a somehow disturbing columns-of-pixels effect

```
   b) FullRes: set BPLCON1 to 0

5. set the COLORxx registers (xx ranges from 0 to 15 or, if MskPln active,
   to 31) to the values found in
               these sections
                or, for
               Cross Playfield
               ,
   in
               this section
                6. initialize a pointer called "ChnkScr" as follows:
   a) FullRes: ChnkScr = address of the homonymous buffer
   b) HalfRes: ChnkScr = VdoPln0




 - given the way BPLCON1 is treated at point 4.a,
               horizontal scrolling
                is
   harder than usual: BPLCON1 can still be used for that purpose, but many
   more complications derive from the different shift value of the play-
   fields and, above all, from the settings required by the
               chequer effect
                 - for
               buffered displays
                and
               Cross Playfield
                some other additional buf-
   fers must be reserved in CHIP and/or FAST ram
```

1.4 RGBx Color Composition

```
                3.3   RGBx Color Composition



You have read somewhere that TCS is based on RGBx color composition: sure-
ly this name has left you a little uncertain, because there's an "alien"
letter queued to the very common RGB initials; that 'x' is a "variable",
because several color composition methods can be constructed and used
under TCS.
In this section you'll learn how to handle them properly:

 3.3.1
                General Information on RGBx Formats
                Here are some possible RGBx formats (all embedded in the tcs. ↩
                   library):

 3.3.2
                RGBW Color Composition
                 3.3.3
                RGBW Palette Settings
                 3.3.4
                RGBW <-> RGB Conversion
```

1.5 3.3.1 General Information on RGBx Formats

3.3.1 General Information on RGBx Formats

This section covers the common aspects among the various RGBx formats.

1.6 3.3.1.1 WarmUp

3.3.1.1 WarmUp

The fundamental staring point is: RGBx color composition works on the same basis of normal RGB in the sense that a color is given by the composition of several (4) sub-components.
That's why the first thing we must do is clearly define the RGBx components, their abbreviations and meanings:

CN	CID	color
RN	0	"red"
GN	1	"green"
BN	2	"blue"
xN	3	<format-dependent>

where: CN=Component Name; CID=Component ID

From these definitions we can build a little vector CCID[], indexed by CNs, which returns the CID of the components:

```
CCID[RN] = 0
CCID[GN] = 1
CCID[BN] = 2
CCID[xN] = 3
```

and a similar vector Ccol[] which returns the color:

```
Ccol[RN] = "red"
Ccol[GN] = "green"
Ccol[BN] = "blue"
Ccol[xN] = <format-dependent>
```

Don't be afraid! These are *not* real vectors! We will use them only in this doc anytime we need to be a bit more precise than natural words.

Meaning of some (frequent) senteces:

- "[the component] RN" = "the component commonly indicated by 'R' or 'red'"
- "the component with CID=1" = "the component commonly indicated by 'G'"
- "the component #CID" = "the component CN with CCID[CN]=CID"
- "B" = "the intensity of the component BN" = "the intensity of the component indicated by/marked as 'B'" = a number in the range [0...255]
- "the CID of the component xN" = CCID[xN] = 3
- "the color of the component RN" = Ccol[RN] = "red"

Urgh! It seems sooooooooo silly... but, please *do* believe me, they are necessary to avoid a lot of mess when things get complicated!

Some more nomenclature about component intensities:

```
C =   0 = "null" or "black"
C =  85 = "dark"
C = 127 = "half"
C = 170 = "dimmed"
C = 255 = "full"
```

1.7 3.3.1.2 Bits Allocation Inside RGBx Pixels

3.3.1.2 Bits Allocation Inside RGBx pixels

As anticipated early in the introduction part, each pixel on the screen consists in a byte in ChnkScr. Since we're studying a TrueColor-ish video system, the bits inside such byte represent the RGBx components of the color of the pixel.
The most general allocation of those bits is:

```
bit #      7  6  5  4  3  2  1  0
bit name   R1 G1 B1 X1 R0 G0 B0 X0
```

For now let's forget about the Xns.

Each of these components, being represented on 2 bits, can range from 0 to 3:

```
  RV       GV       BV

 R1 R0    G1 G0    B1 B0    C

 0  0     0  0     0  0       0    (null)
 0  1     0  1     0  1      85    (dark)
 1  0     1  0     1  0     170    (dimmed)
 1  1     1  1     1  1     255    (full)
```

pay attention: CV (Component Value) is just another way of indicating the intensity C of the component CN in the range [0...3] (Cn indicates the n-th bit of CV)

As already discussed
, being so close to one another, they are
"confused" and "mixed" by the human eye in the very well known RGB manner.

A full red is obtained with R1=R0=1 and all the rest set to 0: %10001000;
same for green (%01000100) and blue (%00100010);
a dimmed red is obtained with R1=1, R0=0: %10000000;
a dark green is obtained with G1=0, G0=1: %00000100;

Many other colors derive from the combinations of those components;
for instance, if we want the white color, all we have to do is:

```
R=255   ->   RV=3   ->   R1=1, R0=1   >
G=255   ->   GV=3   ->   G1=1, G0=1   > %11101110
```

```
 B=255   ->   BV=3   ->   B1=1, B0=1   >

a dimmed white (a shade of grey) would be obtained with:

 R=170   ->   RV=2   ->   R1=1, R0=0   >
 G=170   ->   GV=2   ->   G1=1, G0=0   > %11100000
 B=170   ->   BV=2   ->   B1=1, B0=0   >

a dark white (another grey) with:

 R=85   ->   RV=1   ->   R1=0, R0=1   >
 G=85   ->   GV=1   ->   G1=0, G0=1   > %00001110
 B=85   ->   BV=1   ->   B1=0, B0=1   >

black, obviously:

 R=0   ->   RV=0   ->   R1=0, R0=0   >
 G=0   ->   GV=0   ->   G1=0, G0=0   > %00000000
 B=0   ->   BV=0   ->   B1=0, B0=0   >

full yellow:

 R=255   ->   RV=3   ->   R1=1, R0=1   >
 G=255   ->   GV=3   ->   G1=1, G0=1   > %11001100
 B=  0   ->   BV=0   ->   B1=0, B0=0   >

full purple:

 R=255   ->   RV=3   ->   R1=1, R0=1   >
 G=  0   ->   GV=0   ->   G1=0, G0=0   > %10101010
 B=255   ->   BV=3   ->   B1=1, B0=1   >

And so on...

Actually, the colors we have found so far aren't equivalent to the RGB
colors obtained with the same assignments to the components.
This is because there is a couple of bits which have their own influence
in the final outcome: the bits we marked with Xn.
With all the possible combination of these bits, many other colors become
available (4 times as much).
The RGBx formats differ from one another in function of how the Xn bits
are treated.
```

1.8 3.3.1.3 How Pixels Are Plotted on a HalfRes Screen

```
                3.3.1.3   How Pixels Are Plotted on a HalfRes Screen


After reading the
               section 3.3.1.2
                you may now wonder how the Cn bits are
handled and put together to generate the proper colors: everything is fa-
irly simple and does not require any CPU, Blitter or Copper intervention.
Already guessed how thanks to the techie hints in
```

```
               section 3.2
               ?
Great!...
... but let's talk about it anyway...
```

At 1st glance, it may seem strange that BPL1PT and BPL2PT both point to the same memory area (VdoPln0): OK, strange but not wrong. Notice that playfield 2 (bitplane 2) is scrolled horizontally by 1 LORES pixel, which corresponds exactly to 4 SHRES pixels: this means that the low-order nibble of the RGBx byte in bitplane 1 is "covered" by the hi-order nibble of the same byte in bitplane 2, as shown in the following diagram:

```
 plane 2               R1 G1 B1 X1 R0 G0 B0 X0
 plane 1   R1 G1 B1 X1 R0 G0 B0 X0
                       ^^^^^^^^^^^
                      1 LORES pixel
```

The Amiga's planar system automatically puts C0 and C1 together to generate an index to the color registers, so we need not doing anything!

The sharp-eyed readers surely have already noted that all this is *not* enough to generate and keep separate all the RV, GV, BV and XV values: in fact, with just 2 planes can map only one of them.
Indeed we need 16 combinations (4 for each component), that's why the list of

```
               section 3.2
                includes a couple of "selector planes".
```

The 2 lowest bits of the index are the C1 and C0 bits and select the intensity, the other have to be provided by the "selector planes" to select the component: according to the values of the CIDs, the patterns the SclPlns have to be filled with are:

```
 component selected :     R G B x   R G B x

 SlcPln1 pattern    :   % 0 0 1 1   0 0 1 1   =   %00110011
 SlcPln0 pattern    :   % 0 1 0 1   0 1 0 1   =   %01010101

                          ^ ^ ^ ^   ^ ^ ^ ^
                          | | | |   idem
                          | | | |
                          | | | CCID[x]
                          | | CCID[B]
                          | CCID[G]
                          CCID[R]
```

A scheme for the RGB palette to use for *our* display would look more or less like this:

COLORxx -$dff180	SlcPlns value	CV	R	G	B	
0	%00	%00	0	0	0	(null red)
1	%00	%01	85	0	0	(dark red)
2	%00	%10	170	0	0	(dimmed red)
3	%00	%11	255	0	0	(full red)
4	%01	%00	0	0	0	(null green)

```
 5          %01       %01     0  85   0   (dark green)
 6          %01       %10     0 170   0   (dimmed green)
 7          %01       %11     0 255   0   (full green)

 8          %10       %00     0   0   0   (null blue)
 9          %10       %01     0   0  85   (dark blue)
 10         %10       %10     0   0 170   (dimmed blue)
 11         %10       %11     0   0 255   (full blue)

 12         %11       %00   *** *** ***   (these values indicate the color
 13         %11       %01   *** *** ***   of the x component, so they have
 14         %11       %10   *** *** ***   to be defined with each the RGBx
 15         %11       %11   *** *** ***   specific definition)
```

in fact, the component GN of a green pixel (%01000100) is generated by:

```
  plane #   plane name   value

  4         SlcPln1      %00110011
  3         SlcPln0      %01010101
  2         VdoPln0      %    01000100
  1         VdoPln0      %01000100
                              ^
                              |
                              +--- %0111 = 7
```

Please note that SlcPlns effectiveness is in no way affected by the BPLCON1 scroll (if defined as in Amiga Setup section).

Unfortunately, the LORES pixels surrounding the one we wanted are affected as well, so that two consecutive pixels are separated by another pixel. We could say that each pixel is actually 2 LORES pixels wide (our resolution is 160x256 with a pixel ratio of 2:1), but this would be a bit reductive.
With a slightly deeper analisys, we notice that the pixel in the middle is a sort of "average" of the surrounding pixels.
This is better explained by expanding the previous diagram:

```
 plane 2               R1 G1 B1 X1 R0 G0 B0 X0 r1 g1 b1 x1 r0 g0 b0 x0
 plane 1   R1 G1 B1 X1 R0 G0 B0 X0 r1 g1 b1 x1 r0 g0 b0 x0
                       ^^^^^^^^^^^ ^^^^^^^^^^^ ^^^^^^^^^^^
                       pixel1 (p1)  "average"  pixel2 (p2)
                                      pixel
                                      (1a2)
```

As you can see, 1a2 can be considered an "average" of p1 and p2 because its components are made of both p1's and p2's.
This could turn out to be a nice side-effect, but, since in 1a2 the Cn "weights" are exchanged (R0 has greater influence than r1), generally it represents a problem.
For example, consider a dimmed red pixel (%10000000) in a completely black field: R1 "activates" a dark red dot on the left:

```
 plane 2           1 0 0 0 0 0 0 0
 plane 1   1 0 0 0 0 0 0 0
```

```
           ^^^^^^^ ^^^^^^^
            dark   dimmed
             red     red
            pixel   pixel
            (85)    (170)
```

which can be accepted like a "blur" or "anti-alias" effect.
Also the %10001000 case presents a similar situation:

```
 plane 2           1 0 0 0 1 0 0 0
 plane 1   1 0 0 0 1 0 0 0
           ^^^^^^^ ^^^^^^^ ^^^^^^^
            dark    full   dimmed
             red     red     red
            pixel   pixel   pixel
            (85)    (255)   (170)
```

But we're not so lucky in the %00001000 situation:

```
 plane 2           0 0 0 0 1 0 0 0
 plane 1   0 0 0 0 1 0 0 0
           ^^^^^^^ ^^^^^^^ ^^^^^^^
            black   dark   dimmed
            pixel    red     red
                    pixel   pixel
              (0)    (85)   (170)
```

This means that the contribution to the "average" on the right side is greater than the color iteself, which is quite bad.
The only way I can see to get around this is to activate another plane which either masks out the "average" pixels (its name would be "MskPln") or, even better, "re-inverts" the "importance" of Cns in the "average" pixels.
The pattern MskPln has to be filled with is %11110000, where 0s are neutral and 1s affect the "average" pixels only (the 1-0 order is due to the fact that MskPln belongs to the same playfield of VdoPln0 and so has the same scroll shift).
When the MskPln is activated, to "re-invert" the Cns in the "average" pixels, the palette must be extended in this way (to mask them out just set all the following to 0):

COLORxx -$dff180	MskPln_ SlcPlns value	CV	R	G	B	
16	%1_00	%00	0	0	0	(null red)
17	%1_00	%01	170	0	0	(dimmed red)
18	%1_00	%10	85	0	0	(dark red)
19	%1_00	%11	255	0	0	(full red)
20	%1_01	%00	0	0	0	(null green)
21	%1_01	%01	0	170	0	(dimmed green)
22	%1_01	%10	0	85	0	(dark green)
23	%1_01	%11	0	255	0	(full green)
24	%1_10	%00	0	0	0	(null blue)
25	%1_10	%01	0	0	170	(dimmed blue)

```
 26         %1_10     %10     0   0  85   (dark blue)
 27         %1_10     %11     0   0 255   (full blue)

 28         %1_11     %00   *** *** ***   (these values indicate the color
 29         %1_11     %01   *** *** ***   of the X component, so they have
 30         %1_11     %10   *** *** ***   to be defined with each specific
 31         %1_11     %11   *** *** ***   RGBx definition)
```

This example shows how it works in case of a dark red pixel near to a dimmed green one:

```
 plane #   plane name   value

 5         Mskpln       %11110000 11110000
 4         SlcPln1      %00110011 00110011
 3         SlcPln0      %01010101 01010101
 2         VdoPln0      %    0000 10000100 0100
 1         VdoPln0      %00001000 01000000
                             ^^^^     ^^^^
                                  ^^^^
                             dark avg dimmed
                              red     green
```

If the Cns weren't re-inverted, the "average" pixel would have a dimmed red component and a dark green one. Instead we need exactly the opposite: in fact the red and green components of the "average" pixel are, respectively, %10010 (= 18 = dark red) and %10101 (= 21 = dimmed green).
Yet, we still haven't got rid of all problems; consider the situation below:

```
 plane #   plane name   value

 5         Mskpln       %11110000 11110000
 4         SlcPln1      %00110011 00110011
 3         SlcPln0      %01010101 01010101
 2         VdoPln0      %    0000 10001000 0000
 1         VdoPln0      %00001000 10000000
                             ^^^^     ^^^^
                                  ^^^^
                             dark avg dimmed
                              red     red
```

What?!? The avg color results brighter than the ones which surround it!!!
In fact, we have that the dark and dimmed Rns combined to give a full red!
The solution to this problem could be darkening all or some of the RGB values of colors 16-31 (for example, all the full colors in that range could have their full components halved to 128 - this causes a certain loss of brightness);
but this, on the other side, wouldn't give a good "average" in case of consecutive pixels with full colors...

...the choice must be done carefully, so let's make it the "analytic" way; we have two pixels (R1G1B1X1R0G0B0X0 and r1g1b1x1r0g0b0x0) attached:

```
 plane #   plane name   value

 2         VdoPln0                 R1 G1 B1 X1 R0 G0 B0 X0
```

```
 1          VdoPln0      R1 G1 B1 W1 R0 G0 B0 X0 r1 g1 b1 x1 r0 g0 b0 x0
                                                  ^^^^^^^^^^^
                                                  ^^  avg
                                                  RV'
```

For each component (except x, which will be discussed in relation to any RGBx format), we list the intensities Cs and cs that generate all the 4 possible CV's:

```
  CV'

 C0 c1   C     c           how C and c have been found

 0  0      0     0         C0=0 -> CV=(%00 or %10) -> C=(  0 or 170)
         170    85         c1=0 -> cv=(%00 or %01) -> c=(  0 or  85)

 0  1      0   170         C0=0 -> CV=(%00 or %10) -> C=(  0 or 170)
         170   255         c1=1 -> cv=(%10 or %11) -> c=(170 or 255)

 1  0     85     0         C0=1 -> CV=(%01 or %11) -> C=( 85 or 255)
         255    85         c1=0 -> cv=(%00 or %01) -> c=(  0 or  85)

 1  1     85   170         C0=1 -> CV=(%01 or %11) -> C=( 85 or 255)
         255   255         c1=0 -> cv=(%10 or %11) -> c=(170 or 255)
```

But, since C and c can be mixed in any combination (inside each CV' sub-class), we have the following table:

```
      combination
 CV'   C  -  c      ideal RGB average ( (C+c)/2 )

 0      0 -   0       0
        0 -  85      43
      170 -   0      85
      170 -  85     128

 1      0 - 170      85
        0 - 255     128
      170 - 170     170
      170 - 255     213

 2     85 -   0      43
       85 -  85      85
      255 -   0     128
      255 -  85     170

 3     85 - 170     128
       85 - 255     170
      255 - 170     213
      255 - 255     255
```

It seems sensible to assign to C' the intensity calculated as the average of its ideal averages (approximate/idealized somewhere...):

```
 CV'  C'

 0      0 +
```

```
       43 +
       85 +
      128 = 256   -> [/4] ->    64

 1     85 +
      128 +
      170 +
      213 = 596   -> [/4] ->   149

 2     43 +
       85 +
      128 +
      170 = 426   -> [/4] ->   107

 3    128 +
      170 +
      213 +
      255 = 766   -> [/4] ->   192
```

It's a pleasure to see that these values are "re-inverted" as we intuitively supposed;
let's put them in the palette table:

COLORxx -$dff180	MskPln_ SlcPlns values	CV'	C'
16+4*CID	%1_CCID[CN]	00	64
17+4*CID	%1_CCID[CN]	01	149
18+4*CID	%1_CCID[CN]	10	107
19+4*CID	%1_CCID[CN]	11	192

the values of components different from CN must be set to 0.

- another side effect related to "average" pixels is tackled with equal shrewdness in
  section 3.4

1.9 3.3.1.4 How Pixels Are Plotted on a FullRes Screen

3.3.1.4 How Pixels Are Plotted on a FullRes Screen

Luckily most of what's been discussed about in the
HalfRes section
holds
true also in FullRes. There are several significant differences, though.

Unfortunately here we cannot start with "... you may now wonder how the Cn bits are handled and put together to generate the proper colors: everything is fairly simple and does not require any CPU, Blitter or Copper in-

tervention..." like we did before. Instead, we have that the CPU (and, possibly, the Blitter) must make a huge effort to keep things going fast.

OK, let's proceed by degrees.

FullRes, as its name (opposed to HalfRes) suggests, has a horizontal resolution of up to 320 pixels per line. Let's cut&paste a piece from the

previous section
to see why HalfRes can't reach such limit:

```
 plane 2               R1 G1 B1 X1 R0 G0 B0 X0
 plane 1   R1 G1 B1 X1 R0 G0 B0 X0
                       ^^^^^^^^^^^
                     1 LORES pixel
```

In this cut-out the bits [R1 G1 B1 X1] and [R0 G0 B0 X0] "brim over" and brutally "invade the neighbourhood", forming the well-dissected "average" pixels: the resolution is halved because there is one of them for each "normal" pixel.

This means that we have to find a way to get rid of those "average" pixels. It's not a case that this chapter has been opened with a hint to some heavy CPU work: I can't really imagine a way of doing such operation by resorting just to the Amiga's video circuitry.
One big problem is that the final video data must reside in CHIP ram (in order to be fetchable by the aforementioned hardware) which happens to be deadly slow for today's standards and tasks similar to the one we are facing now: so WE CAN'T EXPECT THIS VIDEO MODE TO BE BLISTERING FAST, especially if compared to HalfRes.

Despite of this, we, in the "Pure Amigan" spirit, don't throw our hopes away, confident that our marvelous machine can actually handle this situaton.

For a start, let's say that although we have seen many miracles on the Amiga, we won't get any far without a good quantity of FAST ram: even if we can parallelize CHIP ram writes with internal CPU processing, we can't do it on reads (at least on non-superscalar CPU's... in other words: maybe only the 060 can partially overcome this limitation... but I dare you to find somebody who owns a 060 accelerator without FAST ram!).
Indeed, FullRes is possible on unexpanded A1200s, but, without surprise, the resulting speed is terrible.

We are ready now to start with the techie part.
Following the

setup section
directions, we:

- allocate two 1280x256 planes, called VdoPln0 and VdoPln1, in CHIP ram
- allocate a 320x256 (=10240 bytes) buffer (possibly in FAST ram) called ChnkScr
- load the BPLxPT registers with:

```
BPL4PT   SlcPln1
BPL3PT   SlcPln0
```

```
    BPL2PT   VdoPln1
    BPL1PT   VdoPln0
```

ChnkScr is the buffer from/to which we read/write the pixels in chunky fashion, while VdoPln0 and VdoPln1 are 2 separate buffers read by the bitplane DMA and shown on the screen.

Now it's possible to give an algorithmic description of the process to execute.
Going back to the problem that originated this discussion, we can clearly see that the CPU must perform this job:

1. fetch the data (pixels) from ChnkScr
2. convert the data
3. write the data to both VdoPln0 and VdoPln1

The conversion consists in separating the nibbles which the pixels are made of and writing them to their own plane:

```
 ChnkScr   ... R1 G1 B1 X1 r1 b1 g1 x1 R2 G2 B2 X2 r2 g2 b2 x2 ...

               ^         ^ ^         ^  ^                     ^
               |         | |         |  |                     |
               |         | |         |  +---------------------+---pixel2
               |         | |         |
  p1.nibble1---+---------+ +---------+---p1.nibble0

               ^                     ^
               |                     |
               +---------------------+---pixel1
```

pixel1 can be seen as the concatenation of 2 nibbles:

- p1.nibble1: most-significant bits of pixel1's CVs
- p1.nibble0: least-significant bits of pixel1's CVs

It's immediate to see that for each pixel Y of ChnkScr pY.nibbleX should be written to VdoPlnX, in order to have the data organized in this way:

```
 VdoPln1   ... R1 G1 B1 X1 R2 B2 G2 X2 ...
 VdoPln0   ... r1 g1 b1 x1 r2 b2 g2 x2 ...

               ^         ^ ^         ^
               |         | |         |
      pixel1---+---------+ +---------+---pixel2
```

In a nutshell: the pixels halves must be adequately arranged in "columns" on the VdoPlns.

In practice this is exactly what HalfRes does by using the same plane for VdoPln0 and VdoPln1 and shifting the playfields by 1 LORES pixel.
The great difference, here, is that there are no longer bits bursting out; the price is that the CPU is occupied by a heavy conversion loop.

A little help can be obtained from the Blitter, which can take some of the conversion load at the cost of an additional buffer in CHIP memory; here's

what we can do:

for maximum speed we fetch source data from FAST ram by longwords; thus each longword holds 4 chunky pixels in this format: $1a2b3c4d, where $1a is the first, $2b is the second, $3c the third and $4d the fourth pixel. Our goal is to write $1234 to VdoPln1 and $abcd to VdoPln0.
To minimize the CPU job, we turn $1a2b3c4d into $1a3c2b4d (which requires just 3 rotate instructions) and write it to the additional buffer in CHIP memory (by arranging properly fetches/writes/conversions, conversions can be done in parallel with the slow writes to CHIP ram).
After writing the whole buffer, the Blitter can convert the data for VdoPln1 in this way:

	data	shift (R)	modulo (+)
channel A:	$1a3c	0	+2
channel B:	$2b4d	4	+2
channel C:	$f0f0	-	-2
	V		
channel D:	$1234	-	0

(D data is given by: (A & C) | (~A & B) - this is just one of the several equivalent ways of doing this)

VdoPln0 can be similarly obtained by blitting in descending mode:

	data	shift (L)	modulo (-)
channel A:	$1a3c	4	+2
channel B:	$2b4d	0	+2
channel C:	$f0f0	-	-2
	V		
channel D:	$abcd	-	0

Note that each blit is just 1 word wide, so BLTSIZV must be ScrWd*ScrHt/4 (the /4 comes from the width and the modulo of the blit; at most we can convert ScrHt=(4*MaxBLTSIZV)/ScrWd lines: for a 320 pixels wide screen it is (32768*4/320)=409 lines).

This technique is generally useless on machines without FAST ram, as its main purpose is to let the CPU free to work in FAST while the Blitter in parallel executes the FullRes conversion.

Now that all the main actors have made their appearance, we have to do justice to a few extras who preferred staying hidden all this time:

"average" pixels totally disappeared therefore MskPln and ChqrMode are not needed anymore; this translates directly into three facilitations:

```
- bitplane and Copper DMA transfers are reduced, so the CPU enjoys a
  better access to CHIP ram (if
             Blitter-assisted conversion
              isn't used)
- only 16 COLORxx registers have to be loaded, so the Copper executes
  less instructions than it did with MskPln activated (and the CPU has
  less writes to the copperlist when the palette is changed for fade-fx
  and the likes)
- using BPLCON1 for horizontal
             scrolling
              remains as easy as it's ever
  been in Amiga's history (yet, it can be much more simply achieved by

             selecting only certain parts of ChnkScr
             , as the TCS_CPUFRPass1()
  and TCS_CPUFRPass2() of tcs.library permit to do)
- if you need
             screen buffering
              only to avoid on-screen jerkings, you
  can completely forget about it if CPU-only FullRes conversion is used:
  in fact, both VdoPlns are updated at the same time longword by long-
  word, so no visual side-effect appears on screen (this, instead, does
  not happen with
             Blitter-assisted conversion
              because it is carried
  out on planar basis)
```

1.10 3.3.1.5 RGB <-> RGBx Conversion

```
                3.3.1.5   RGB <-> RGBx Conversion
```

One of the great problems that an RGBx format presents is that they are unsupported by the common paint packages (I don't know if any RGBx-like encoding has been already defined and adopted somewhere; I'm skeptical because RGBx is way too weird...), with the consequent difficulty in using any kind of pre-existent picture.

Finding an automatic method which makes using images fast, comfortable and (above all!) possible is the target of this section.

Let's look at an RGB palette like a vector of this kind:

```
 PAL = array [0...255] of record
                           R: 0...255
                           G: 0...255
                           B: 0...255
                          endrecord
```

Calling, for simplicity, PAL[CL] the 24-bit value obtained by the concatenation of PAL[CL].R, PAL[CL].G and PAL[CL].B, it would be marvelous if CL was exactly the RGBx value which, on a Tricky-Color display, looks like PAL[CL] on a normal screen.

In other words, we could say that the vector PAL[] is RGBx-indexed.
In that way we could remap any picture to the palette PAL[] (evidently this could worsen the picture's quality) with a normal image-processing package and then save it in raw 8-bit chunky format, to have it ready to be shown on a Tricky-Color screen!
So our task consists in finding an algorithm which generates PAL[].

Now we get even more empirical, but, since the RGB definition itself seems to have an empirical basis, let's not be too fussy.

Each color comes with a luminance signal, which, consequently, is a function of each component of the associated color:

```
 Luminance in RGB = Lrgb(R,G,B) = Lr*R + Lg*G + Lb*B
```

Working in 24-bit:

```
 R,G,B and Lrgb() all belong to the range [0...255];
 Lrgb(R,G,B) = 0     <=>   R=G=B=0;
 Lrgb(R,G,B) = 255   <=>   R=G=B=255;

 therefore necessarily:   Lr+Lg+Lb = 1.0;
```

Intuitively, the Lns tell how much each single component affects the total luminance.
Sperimentally, (ages ago) it has been found that:

```
 Lr = 0.299
 Lg = 0.587
 Lb = 0.114
```

The same should happen in the RGBx format:

```
 Luminance in RGBx = Lrgbx(R,G,B,X) = Lr'*R + Lg'*G + Lb'*B + Lx'*X

 where Lr'+Lg'+Lb'+Lx' = 1.0;
```

The problem is that the normal RGB values of Lns no longer hold, due to the addition of the 4th component.
Here comes the empirical, arbitrary part: with a little of guess-working and/or brain-driven testing one has to find an acceptable value to assign to Lx', to throw away the "exceeding" unknown.
Since PAL[] sensibly changes accordingly to Lx', it must be chosen bearing in mind 2 criteria:

- searching for the PAL[] which gives the best _final_ display result (which does *not* necessarily mean the best-looking PAL[] in absolute)
- trying to get the highest number of _unique_ colors possible (being a calculated palette, it could happen that PAL[i]=PAL[j], i<>j)

To start we must note that there is a little flaw in the considerations about the luminance: an RGBx pixel is actually made up of 4 pixels, so, theorically, each of these sub-pixels affects the total luminance by 1/4. However, this would be true only if each sub-pixel was a gray-shade; instead, since a sub-pixel has only one component, we lose some brightness. Thus the contribution of each sub-pixel is:

```
 RN -> Lr*0.25
 GN -> Lg*0.25
 BN -> Lb*0.25
 (the contribution of xN, <= 0.25, depends on the RGBx mode)
```

By definition Lr+Lg+Lb = 1, so the first 3 sub-pixels only give 1/4 of the max luminance possible; in other words, since xN at most contributes with 1/4, there is always a loss of brightness of at least 50%.
Being this just a side-effect of the way pixels are displayed on the monitor, in the following calculations we won't care about it; these information will be useful when calculating the specific Lx's values.

Supposing to have found the Lx' value:

```
 Lr'+Lg'+Lb'+Lx' = 1.0   ->   Lr'+Lg'+Lb' = 1.0-Lx'
```

to determine the remaining Ln's we can observe that surely they still are directly proportional to the respective Lns; in fact, the previous equation can be decomposed as:

```
 Lr'+Lg'+Lb' = 1.0-Lx'   ->   Lr'+Lg'+Lb' = (1.0-Lx')*(Lr+Lg+Lb)   ->

      Lr' = (1.0-Lx')*Lr
 ->   Lg' = (1.0-Lx')*Lg
      Lb' = (1.0-Lx')*Lb
```

Now, according to these definitions, we can say that a color that looks the same in both RGB and RGBx formats has also the same luminance:

```
 Lrgb(R,G,B) = Lrgbx(R',G',B',X')
```

Remeber what we are trying to achieve: we have to build an RGB palette starting from all the possible RGBx values.
We have those values, because in an 8-bit palette they are simply the indexes between 0 and 255. Thus, in the previous equation, the unknowns are the R, G and B values in Lrgb()!

At this point we can observe that one generical component CN contributes to Lrgb() exactly as much as the corresponding CN' component *and sometimes* (part of) XN' contribute to Lrgbx().
I'm not sure at all of these assumptions, but they succed in real-world tests!

Different RGBx formats are characterized by different values of Lx' and different ways in which X' affects Lrgbx().

We now write a useful (will come in handy later!) function which, given a chunky pixel in RGBx format and a CID, returns the intensity of the component CN such that CCID[CN]=CID:

```
 function GetIntensity(RGBxPxl,CID)

  ;RGBxPxl = chunky pixel in RGBx format
  ;CID     = Component ID as defined in
              section 3.3.1.1
               (CCID[y])
  ;note    : for CID=CCID[x]=3 the result could be meaningless depending
```

```
             on the RGBx format

  intensity=0
  if <bit no. 3-CID of RGBxPxl is 1> then intensity=intensity+85
  if <bit no. 7-CID of RGBxPxl is 1> then intensity=intensity+170

 return[intensity]
```

Looking at these considerations from another point of view, we can say that we have just found a method for converting 8-bit RGBx values to normal 24-bit RGB ones, which is useful for remapping pictures with any palette.

We can additionally ask ourselves: what if we wanted to do exactly the opposite (i.e. converting from normal TrueColor 24-bit to RGBx)?
The problem is quite hard and it's impossible to give a complete description here because the conversion is RGBx-format dependant, so all the necessary information will be given in each mode specific section.
A common problem, instead, that we'll have to face when doing this kind of conversion is the reduction from 8-bit components to 2-bit ones, so we'd better write another handy function.

For any 8-bit RGB component, the best matching one among the 4 available in RGBx must be found:

```
 RGB    {0,.....,255}

         |    |    |
         V    V    V

 RGBx   {0,85,170,255}
```

How to do this choice? Well, "best-matching value" can be thought as "the closest value"; considering that RGBx values grow with a "step" of 85, it is quite easy to see that the RGB set can be divided in the following subsets:

```
 {0,...,42}, {43,...,85,...,127}, {128,...,170,...,212}, {213,...,255}
  *                 *                       *                      *
```

so, converting the 8-bit component CN is reduced to picking the value marked with "*" that appears in the subset which C belongs to.
Indeed, in what we need is not the component intensity, but the 2-bit Component Value:

```
   C    CV

   0 -> %00
  85 -> %01   ->   CV = (C+42)/85
 170 -> %10
 255 -> %11
```

So our function will simply be: C2CV(C) = (C+42)/85

- when remapping a picture using PAL[], dithering helps a lot in most of

```
   the cases (anyway, judge from the final output quality on a TCS screen,
   *not* from how it looks just after conversion!)
 - the function TCS_MkRGBxCnvTab() of the tcs.library provides a simple
   means of producing the PAL[] array
```

1.11 3.3.2 RGBW Color Composition

```
3.3.2   RGBW Color Composition


The concept behind this format is that the colors deriving from the RGB
composition can form new tonalities by altering their brightness.

Here is the specific RGBW bits allocation:

 bit #      7  6  5  4  3  2  1  0
 bit name   R1 G1 B1 W1 R0 G0 B0 W1

We decide to use the extra bit to generate the white component (hence the
"W"), which can be seen as a "contribution" to the brightness of the other
components.
It is exactly treated like the other ones, with the only difference that
W now it's not a simple intensity, but a real color:

  WV

 W1 W0   W

 0  0      0   0   0   (null white)
 0  1     85  85  85   (dark white)
 1  0    170 170 170   (dimmed white)
 1  1    255 255 255   (full white)

The intensity of W is exactly the value of its components, so the little
abuse of the terminology is partially justified.
With this in mind, we can now say that the brightest white possible is:

 R=255   ->   RV=3   ->   R1=1, R0=1   >
 G=255   ->   GV=3   ->   G1=1, G0=1   > %11111111
 B=255   ->   BV=3   ->   B1=1, B0=1   >
 W=255   ->   WV=3   ->   W1=1, W0=1   >

 a simple %11101110 would be much less brighter and look grey-ish.

In the same way we can obtain cyan with:

 R=  0   ->   RV=0    ->   R1=0, R0=0   >
 G=  0   ->   GV=0    ->   G1=0, G0=0   > %00110011
 B=255   ->   BV=3    ->   B1=1, B0=1   >
 W=255   ->   WV=3    ->   W1=1, W0=1   >

And so on...
```

1.12 3.3.3 RGBW Palette Settings RGBW

```
                3.3.3   RGBW Palette Settings RGBW


(the color settings listed here don't include the ones already specified
in
               section 3.3.1.3
               )

The WN component must be white:

 COLORxx    SlcPlns        +---W---+
 -$dff180   values    WV   R   G   B

 12         %11       00     0   0   0   (null white)
 13         %11       01    85  85  85   (dark white)
 14         %11       10   170 170 170   (dimmed white)
 15         %11       11   255 255 255   (full white)

If the MskPln is activated, the "non-darkening" settings for the "average"
pixels would be:

            MskPln_        +---W---+
 COLORxx    SlcPlns
 -$dff180   values    WV   R   G   B

 28         %1_11     00     0   0   0   (null white)
 29         %1_11     01   170 170 170   (dimmed white)
 30         %1_11     10    85  85  85   (dark white)
 31         %1_11     11   255 255 255   (full white)

Otherwise we could find more appropriate values analitically;
we have two pixels (R1G1B1W1R0G0B0W0 and r1g1b1w1r0g0b0w0) attached:

 plane #   plane name   value

 2         VdoPln0                  R1 G1 B1 W1 R0 G0 B0 W0
 1         VdoPln0      R1 G1 B1 W1 R0 G0 B0 W0 r1 g1 b1 w1 r0 g0 b0 w0
                                                ^^^^^^^^^^^
                                                    avg  ^^
                                                         WV'

The Ws and ws that produce all the 4 possible WV's ("dark" = "dark white";
"dimmed" = "dimmed white") are listed in this table:

 WV'  W0 w1   W       w

 0    0  0    black   black
             dimmed  dark
 1    0  1    black   dimmed
             dimmed  white
 2    1  0    dark    black
             white   dark
 3    1  1    dark    dimmed
```

```
              white   white
```

But, since W and w can be mixed in any combination (inside each WV'
sub-class), we have the following table:

```
      combination
 WV'      W-w          ideal RGB average

 0    black-black        0   0   0
      black-dark        43  43  43
      dimmed-black      85  85  85
      dimmed-dark      128 128 128

 1    black-dimmed      85  85  85
      black-white      128 128 128
      dimmed-dimmed    170 170 170
      dimmed-white     213 213 213

 2    dark-black        43  43  43
      dark-dark         85  85  85
      white-black      128 128 128
      white-dark       170 170 170

 3    dark-dimmed      128 128 128
      dark-white       170 170 170
      white-dimmed     213 213 213
      white-white      255 255 255
```

Being W' a real color (rather than a simple component), it seems sensible
assigning to it the color calculated as the average of the ideal averages
(approximate/idealized somewhere...):

```
 WV'  RGB average

 0      0   0   0 +
       43  43  43 +
       85  85  85 +
      128 128 128 = 256 256 256   -> [/4] ->    64  64  64

 1     85  85  85 +
      128 128 128 +
      170 170 170 +
      213 213 213 = 596 596 596   -> [/4] ->   149 149 149

 2     43  43  43 +
       85  85  85 +
      128 128 128 +
      170 170 170 = 426 426 426   -> [/4] ->   107 107 107

 3    128 128 128 +
      170 170 170 +
      213 213 213 +
      255 255 255 = 766 766 766   -> [/4] ->   192 192 192
```

Putting those values in the palette table:

```
            MskPln_         +---W'--+
```

```
 COLORxx    SlcPlns
 -$dff180   values    WV'   R   G   B

 28         %1_11     00     64  64  64
 29         %1_11     01    149 149 149
 30         %1_11     10    107 107 107
 31         %1_11     11    192 192 192
```

1.13 3.3.4 RGBW <-> RGB Conversion

```
                3.3.4   RGBW <-> RGB Conversion


Let's begin from where we left in the
               general part
               :

 Lrgb(R,G,B) = Lrgbx(R',G',B',X')

which in our case can be instanced as:

 Lrgb(R,G,B) = Lrgbw(R',G',B',W')

We have to choose a value for Lw' now; luckily it's very easy to derive
analytically: we know that WN' is a gray shade, so it affects the real
brightness of the pixel exactly by 1/4; recalling that the other three
components affect the real brightness by 1/4, we have that the 50% of
the brightness comes from WN', and the rest from the other components:

 Lw' = 0.5
 Lr' = (1.0-Lw')*Lr = 0.1495
 Lg' = (1.0-Lw')*Lg = 0.2935
 Lb' = (1.0-Lw')*Lb = 0.0570

W' indicates the luminance of the white component WN', which, by defini-
tion, is a shade of gray with 3 sub-components (WN'r, WN'g, WN'b) all of
the same intensity, equal to W' itself (WN'r = WN'g = WN'b = W' - to avoid
confusion, such quantity will be called w'):

 W'   =   Lrgb(w',w',w')   =   Lr*w' + Lg*w' + Lb*w'

When W'>0, all the 3 sub-components are greater than zero: therefore W'
affects all the components we are seeking (this also happens when W'=0:
in that case the influence consists in an automatic darkening as we'll
see).
Examining the components separately:

 Lrgb(R,0,0) = Lrgbw(R',0,0,Lrgb(w',0,0))    ->   Lr*R = Lr'*R' + Lw'*Lr*w'
 Lrgb(0,G,0) = Lrgbw(0,G',0,Lrgb(0,w',0))    ->   Lg*G = Lg'*G' + Lw'*Lg*w'
 Lrgb(0,0,B) = Lrgbw(0,0,B',Lrgb(0,0,w'))    ->   Lb*B = Lb'*B' + Lw'*Lb*w'

in general:

                                          Lc'*C' + Lw'*Lc*w'
```

$$Lc*C = Lc'*C' + Lw'*Lc*w' \quad -> \quad C = \frac{}{Lc}$$

which, written in functional notation (remember that w' = W'):

$$C(C',W') = \frac{Lc'*C' + Lw'*Lc*W'}{Lc} = \frac{Lc'*C'}{Lc} + Lw'*W'$$

that, by substituting Lc', becomes:

$$C(C',W') = (1.0-Lw')*C' + Lw'*W'$$

which can also be read as: the CN component depends directly on the corresponding CN' component, but *also* on the additional contribute by WN': it seems quite sensible, even if the reasoning was contorted (and we can forget about Lr', Lg', and Lb', too).

This may not seem correct, as a pixel value of %10011001 in RGBW format yields an ultra-bright red (pink), not a full red as one could expect:

$$R(255,255) = (1.0-0.5)*255 + 0.5*255 = 255$$

let's not get confused: also the GN and BN components are affected by WN':

$$G(0,255) = (1.0-0.5)*0 + 0.5*255 = 127.5$$

$$B(0,255) = (1.0-0.5)*0 + 0.5*255 = 127.5$$

so the above formula seems to make some sense.
Analogously, a %10001000 cannot be a full red because:

$$R(255,0) = (1.0-0.5)*255 + 0.5*0 = 127.5$$

$$G(0,0) = (1.0-0.5)*0 + 0.5*0 = 0$$

$$B(0,0) = (1.0-0.5)*0 + 0.5*0 = 0$$

As these 2 simple examples show, it's impossible to get a "normal" full red with R=255, G=0, B=0 (this also applies to green and blue, of course).

The final step is writing the algorithm which, making use of the functions C(C',W') and
GetIntensity()
, fills the vector PAL[]:

```
for V=0 to 255

 W' = GetIntensity(V,CCID[W])

 R' = GetIntensity(V,CCID[R])
 PAL[V].R = R(R',W')

 G' = GetIntensity(V,CCID[G])
 PAL[V].G = G(G',W')

 B' = GetIntensity(V,CCID[B])
```

```
  PAL[V].B = B(B',W')

 next V
```

Now we'll deal with the conversion from TrueColor 24-bit to RGBW.
"Reversing" the formula found above:

```
 C = (1.0-Lw')*C'+Lw'*W'
```

to:

```
 C' = (C-Lw'*W')/(1.0-Lw')
```

is pretty useless, because W' is unknown (this is really a catch-22)!
So, we'll have to resort to something that smells a bit like a trick, but still proves to be helpful; first of all, we want the color components to bring the same information of the source:

```
 R' = R,  G' = G,  B' = B
```

However, each value C' is also affected by the presence of WN'; indeed, once fixed C'=C, we don't want WN' to have any influence, so we can intuitively say that W' should not "alter" the brightness given by the other components:

```
 W' = Lr*R'+Lg*G'+Lb*B'
```

The following simple passages show that our intuition is correct;
the general formula (again) tells us:

```
 C = (1.0-Lw')*C'+Lw'*W'
```

from which we find W':

```
 ((Lw'-1.0)*C'+C)/Lw' = W'
```

that, by setting C'=C, becomes:

```
 ((Lw'-1.0+1)*C')/Lw' = W'   ->   Lw'*C'/Lw' = W'   ->   C' = W'
```

but since we have 3, generally different, components, W' cannot be equal to all of them at the same time:

```
  R' = W'
  G' = W'
  B' = W'
```

although the system above can't be satisfied unless the source is a shade of gray, we can operate the following changes:

```
 Lr*R' = Lr*W'
 Lg*G' = Lg*W'
 Lb*B' = Lb*W'
```

that, by summing the terms on both sides, give:

```
 Lr*R'+Lg*G'+Lb*B' = W'*(Lr+Lg+Lb)   ->   Lr*R'+Lg*G'+Lb*B' = W'*1
```

```
which proves the correctness of our little "trick".

Of course, we need to convert all the values found to a Component Value
(2 bits only), so we have to apply a function we wrote
               somewhere before
               :

 CV' =
               C2CV(C')


 - setting Lw' to 0.5 gives only 175 unique colors; 256 unique colors can
   be obtained with a value of ~0.507, but this causes bad conversion of
   some colors
 - RGB -> RGBx conversion isn't very accurate
```

1.14 3.3.5 RGBM Color Composition

```
                3.3.5   RGBM Color Composition


The concept this format is based around is that the Xn bits can be used
to "strengthen" none, 1 or all of the other components.

Here is the specific RGBM bits allocation:

 bit #      7  6  5  4  3  2  1  0
 bit name   R1 G1 B1 M1 R0 G0 B0 M1

Let's focus on the Mns, the Modify bits:

  MV

 M1 M0   Ccol[MN]

 0  0    black
 0  1    full red
 1  0    full blue
 1  1    full white

What does this will ever mean? Simple. The less influencing components
(RN and BN) have now the chance of being "strengthened" by the additional
component MN, whose M ("irregularly") is not a simple intensity, but a
real color with the non-zero RGB components always at 255.

In this way, a strong red would be:

 R=255   ->   RV=3   ->   R1=1, R0=1   >
 G=  0   ->   GV=0   ->   G1=0, G0=0   > %10001001
 B=  0   ->   BV=0   ->   B1=0, B0=0   >
 M=red   ->   MV=1   ->   M1=0, M0=1   >

Equally, a strong blue is given by:
```

```
 R=   0   ->   RV=0   ->   R1=0, R0=0   >
 G=   0   ->   GV=0   ->   G1=0, G0=0   > %00110010
 B= 255   ->   BV=3   ->   B1=1, B0=1   >
 M=blue   ->   MV=2   ->   M1=1, M0=0   >
```

"Uhm... then there must be some sort typo or mistake... probably it was meant to be 'green' instead of 'white' in the table above!" - that's what you're now probably thinking, right?
Well, no. "white" has been intentionally scribbled down there, and now I will tell you why.
Let's imagine we have "green" as you were thinking: can you tell me, now, how on earth one could get the color white?!?
It wouldn't be simply possible.
In fact, once we have set R, G and B all to 255 (and this must be done, otherwise we've already lost the game), we can't choose either of the 4 colors for M, because we need just a white component and nothing else!
(naturally, the whole palette would be affected as well... but this is

```
               another format
               !)
```

Now everything should make sense: we sacrifice the capability of strenghtening the green component (which is already the strongest among the 3), to have the ability of plotting white pixels and have a more "rangeful" palette.

White, like in most modes, is given by %11111111:

```
 R=  255   ->   RV=3   ->   R1=1, R0=1   >
 G=  255   ->   GV=3   ->   G1=1, G0=1   > %11111111
 B=  255   ->   BV=3   ->   B1=1, B0=1   >
 M=white   ->   MV=3   ->   M1=1, M0=1   >
```

1.15 3.3.6 RGBM Palette Settings

```
               3.3.6   RGBM Palette Settings
```

(the color settings listed here don't include the ones already specified in

```
               section 3.3.1.3
               )
```

From the definition given in the

```
               previous section
                follows that the
```

arrangements for the palette are:

```
 COLORxx    SlcPlns        +---M---+
 -$dff180   values    MV   R   G   B

 12         %11       00     0   0   0   (black)
 13         %11       01   255   0   0   (full red)
 14         %11       10     0   0 255   (full blue)
```

```
 15         %11       11   255 255 255   (full white)
```

A new inconvenience pops up, making the use of MskPln compulsory.
Look at this example:

```
 plane #   plane name   value

 5         Mskpln       %11110000 11110000
 4         SlcPln1      %00110011 00110011
 3         SlcPln0      %01010101 01010101
 2         VdoPln0      %    1000 10010011 0010
 1         VdoPln0      %10001001 00110010
                             ^^^^     ^^^^
                                  ^^^^
                           strong avg strong
                              red     blue
```

the "average" pixel is formed not only by the red and blue values of the surronding pixels, but also by a full white component!

It’s quite spontaneous to say: hey! since MV=%11 in the "average" derives from a blue and a red pixels in the surroundings, why don’t we assign it a purple color to the MN component? Sadly, this rather good idea must be discarded because that value is generated also in other ways.
Let’s have a more in-depth look at this.
We have two pixels (R1G1B1M1R0G0B0M0 and r1g1b1m1r0g0b0m0) attached:

```
 plane #   plane name   value

 2         VdoPln0                   R1 G1 B1 M1 R0 G0 B0 M0
 1         VdoPln0      R1 G1 B1 M1 R0 G0 B0 M0 r1 g1 b1 m1 r0 g0 b0 m0
                                                ^^^^^^^^^^^
                                                    avg  ^^
                                                         MV’
```

The Ms and ms that generate all the 4 possible MV’s are listed in this table:

MV’	M0	m1	M	m
0	0	0	black	black
			blue	red
1	0	1	black	blue
			blue	white
2	1	0	red	black
			white	red
3	1	1	red	blue
			white	white

But, since M and m can be mixed in any combination (inside each MV’ sub-class), we have the following table:

MV’	combination MV-mv	ideal RGB average		
0	black-black	0	0	0
	black-red	128	0	0

```
      blue-black      0   0 128
      blue-red      128   0 128

 1    black-blue      0   0 128
      black-white   128 128 128
      blue-blue       0   0 255
      blue-white    128 128 255

 2    red-black     128   0   0
      red-red       255   0   0
      white-black   128 128 128
      white-red     255 128 128

 3    red-blue      128   0 128
      red-white     255 128 128
      white-blue    128 128 255
      white-white   255 255 255
```

It seems sensible to assign to M' the color calculated as the average of the ideal averages (approximate/idealized somewhere...):

```
 MV'  RGB average

 0      0   0   0 +
      128   0   0 +
        0   0 128 +
      128   0 128 = 256 000 256   -> [/4] ->    64   0  64

 1      0   0 128 +
      128 128 128 +
        0   0 255 +
      128 128 255 = 256 256 768   -> [/4] ->    64  64 192

 2    128   0   0 +
      255   0   0 +
      128 128 128 +
      255 128 128 = 768 256 256   -> [/4] ->   192  64  64

 3    128   0 128 +
      255 128 128 +
      128 128 255 +
      255 255 255 = 768 512 768   -> [/4] ->   192 128 192
```

Sperimentally, this solution offers a very good output.
Yet, with this specific choice of colors, we can't obtain "pure" black and white, because, in the "average" pixels, the black-black combination generates a very dark purple (MV'=0) and the white-white combination yields a pale pink (MV'=3). So, at least these 2 cases could be treated separately, by just assigning $00000 and $fffff, respectively. Though, don't forget that these assignments affect also the other combinations in the same class of black (0) and white (3)!!! This choice must be done taking into account the kind of gfx to be displayed.
The values just found are inserted in the palette table below:

```
            MskPln_         +---M'--+
 COLORxx    SlcPlns
 -$dff180   values    MV'   R   G   B
```

```
 28          %1_11      00      64   0  64
 29          %1_11      01      64  64 192
 30          %1_11      10     192  64  64
 31          %1_11      11     192 128 192

or, alternatively, to have "pure" black and "white":

             MskPln_          +---M'--+
 COLORxx     SlcPlns
 -$dff180    values     MV'   R   G   B

 28          %1_11      00      0   0   0
 29          %1_11      01     64  64 192
 30          %1_11      10    192  64  64
 31          %1_11      11    255 255 255



 - mixed "pure" solutions (only one of {black, white} is pure) are also
   possible (colors 28 and 31 are independent!)
 - other "non-pure"-black-and-white (if the "pureness" of those colors
   is not so fundamental) formats are
                RGBS
                 and
                RGBP
```

1.16 3.3.7 RGBM <-> RGB Conversion

```
                3.3.7   RGBM <-> RGB Conversion



Let's begin from where we left in the
                general part
                :

 Lrgb(R,G,B) = Lrgbx(R',G',B',X')

which in our case can be instanced as:

 Lrgb(R,G,B) = Lrgbm(R',G',B',MI'(CID))

 where MI'(CID) = intensity of the component #CID in color M' = 255*Lc'
 (M' couldn't be used directly because in this format it is a real color;
 the fixed value 255 comes from the fact that M' is always a full color,
 except in the case of black, which will be treated appropriately later)

Then, we must choose a seemingly acceptable value for Lm' and calculate
the rest of the Lc's; we must consider that the brightest color of MN is
white, so, similarly to
                RGBW
                , we set Lm' to 0.5 (this introduces a
little error when MN is not white):
```

```
 Lm' = 0.5
 Lr' = (1.0-Lm')*Lr = 0.150
 Lg' = (1.0-Lm')*Lg = 0.293
 Lb' = (1.0-Lm')*Lb = 0.057
```

To calculate the component CN, we can think that its contribution to Lrgb() is equal to the contribution of CN' and *potentially* MN' to Lrgbm(). "Potentially" means that MN' has effect when Ccol[MN']=Ccol[CN'] or when Ccol[MN']=black or Ccol[MN']=white.

```
 Lrgb(R,0,0) = Lrgbm(R',0,0,MI'(0))   ->   Lr*R = Lr'*R' [+Lm'*255*Lr]
 Lrgb(0,G,0) = Lrgbm(0,G',0,MI'(1))   ->   Lg*G = Lg'*G' [+Lm'*255*Lg]
 Lrgb(0,0,B) = Lrgbm(0,0,B',MI'(2))   ->   Lb*B = Lb'*B' [+Lm'*255*Lb]
```

From the formulae above we deduct the general equation:

$$C = \frac{Lc'*C' \ [+Lm'*255*Lc]}{Lc} = \frac{Lc'*C'}{Lc} \ [+Lm'*255]$$

which in functional notation looks like:

$$C(C',MI'(CID)) = \frac{Lc'*C'}{Lc} \ [+Lm'*255] = (1.0-Lm')*C' \ [+Lm'*255]$$

where the [operand] is used only if Ccol[MN']=Ccol[CN'] or when Ccol[MN']=white; this also means that when Ccol[MN']=black the [operand] is omitted and so, being Lc'<Lc, the CN' is always darkened no matter which component it is

To solve the PAL[] problem we just need the C(C',MI(CID)') formula and the function

GetIntensity()

to build a simple algorithm:

```
 for V=0 to 255

  <get M' and MV' from V>              ;used by MI'() and C()

  R' = GetIntensity(V,CCID[R])
  PAL[V].R = R(R',MI'(0))

  G' = GetIntensity(V,CCID[G])
  PAL[V].G = G(G',MI'(1))

  B' = GetIntensity(V,CCID[B])
  PAL[V].B = B(B',MI'(2))

 next V
```

- using a value of 0.5 for Lm' gives 217 unique colors

1.17 3.3.8 RGBS Color Composition

```
3.3.8   RGBS Color Composition



The concept this format is based around is that the Xn bits can be used
to "strengthen" none or 1 of the other components.

Here is the specific RGBS bits allocation:

 bit #      7  6  5  4  3  2  1  0
 bit name   R1 G1 B1 S1 R0 G0 B0 S1

Let's focus on the Sns, the Strengthen bits:

  SV

 S1 S0   Ccol[SN]

 0  0    black
 0  1    full red
 1  0    full green
 1  1    full blue

This means that the component with CID=SV-1 is "strengthened" by the addi-
tional component SN, whose S is not an "intensity" but a real color with
the non-zero RGB components at 255.

In this way, a strong red would be:

 R=255   ->   RV=3   ->   R1=1, R0=1   >
 G=  0   ->   GV=0   ->   G1=0, G0=0   > %10001001
 B=  0   ->   BV=0   ->   B1=0, B0=0   >
 S=red   ->   SV=1   ->   S1=0, S0=1   >

Equally, a strong green is given by:

 R=    0   ->   RV=0   ->   R1=0, R0=0   >
 G=  255   ->   GV=3   ->   G1=1, G0=1   > %01010100
 B=    0   ->   BV=0   ->   B1=0, B0=0   >
 S=green   ->   SV=2   ->   S1=1, S0=0   >

and a strong blue by:

 R=   0   ->   RV=0   ->   R1=0, R0=0   >
 G=   0   ->   GV=0   ->   G1=0, G0=0   > %00110011
 B= 255   ->   BV=3   ->   B1=1, B0=1   >
 S=blue   ->   SV=3   ->   S1=1, S0=1   >

"Pure" white cannot be obtained as the brightest color is:

 R=  255   ->   RV=3   ->   R1=1, R0=1   >
 G=  255   ->   GV=3   ->   G1=1, G0=1   > %11111110
 B=  255   ->   BV=3   ->   B1=1, B0=1   >
 S=green   ->   SV=2   ->   S1=1, S0=0   >
```

1.18 3.3.9 RGBS Palette Settings

```
                3.3.9   RGBS Palette Settings



(the color settings listed here don't include the ones already specified
in
               section 3.3.1.3
               )

From the definition given in the
               previous section
                follows that the
arrangements for the palette are:

 COLORxx    SlcPlns        +---S---+
 -$dff180   values    SV   R   G   B

 12         %11       00     0   0   0   (black)
 13         %11       01   255   0   0   (full red)
 14         %11       10     0 255   0   (full green)
 15         %11       11     0   0 255   (full blue)

This settings cause weird "average" pixels, so we're going to study deeply
what happens.
We have two pixels (R1G1B1S1R0G0B0S0 and r1g1b1s1r0g0b0s0) attached:

 plane #   plane name   value

 2         VdoPln0                  R1 G1 B1 S1 R0 G0 B0 S0
 1         VdoPln0      R1 G1 B1 S1 R0 G0 B0 S0 r1 g1 b1 s1 r0 g0 b0 s0
                                                ^^^^^^^^^^^
                                                    avg  ^^
                                                         SV'

The Ss and ss that generate all the 4 possible SV's are listed in this
table:

 SV'  S0 s1   S       s

 0    0  0    black   black
             green   red
 1    0  1    black   green
             green   blue
 2    1  0    red     black
             blue    red
 3    1  1    red     green
             blue    blue

But, since S and s can be mixed in any combination (inside each SV'
sub-class), we have the following table:

      combination
 SV'       S-s       ideal RGB average
```

```
 0    black-black     0   0   0
      black-red     128   0   0
      green-black     0 128   0
      green-red     128 128   0

 1    black-green     0 128   0
      black-blue      0   0 128
      green-green     0 255   0
      green-blue      0 128 128

 2    red-black     128   0   0
      red-red       255   0   0
      blue-black      0   0 128
      blue-red      128   0 128

 3    red-green     128 128   0
      red-blue      128   0 128
      blue-green      0 128 128
      blue-blue       0   0 255
```

It seems sensible to assign to S' the color calculated as the average of the ideal averages (approximate/idealized somewhere...):

```
 SV'  RGB average

 0      0   0   0 +
      128   0   0 +
        0 128   0 +
      128 128   0 = 256 256   0   -> [/4] ->    64  64   0

 1      0 128   0 +
        0   0 128 +
        0 255   0 +
        0 128 128 =   0 512 256   -> [/4] ->     0 128  64

 2    128   0   0 +
      255   0   0 +
        0   0 128 +
      128   0 128 = 512   0 256   -> [/4] ->   128   0  64

 3    128 128   0 +
      128   0 128 +
        0 128 128 +
        0   0 255 = 256 256 512   -> [/4] ->    64  64 128
```

The values just found are inserted in the palette table below:

```
             MskPln_          +---S'--+
 COLORxx     SlcPlns
 -$dff180    values     SV'   R   G   B

 28          %1_11      00     64  64   0
 29          %1_11      01      0 128  64
 30          %1_11      10    128   0  64
 31          %1_11      11     64  64 128
```

1.19 3.3.10 RGBS <-> RGB Conversion

```
                3.3.10   RGBS <-> RGB Conversion


Let's begin from where we left in the
               general part
               :

 Lrgb(R,G,B) = Lrgbx(R',G',B',X')

which in our case can be instanced as:

 Lrgb(R,G,B) = Lrgbs(R',G',B',S')

Then, we must choose a seemingly acceptable value for Ls' and calculate
the rest of the Lc's; we must consider that the brightest color of SN is
green, so the contribution of this component to the real brightness is
Lg*0.25 (this introduces a little error when SN is not green); we know
also that the remaning components, together, affect the real brightness by
1/4: this means that they have 0.25/(Lg*0.25) = 1/Lg times more influence
than SN, i.e. 1 "part" of the RGBx brightness is given by SN, and the
other 1/Lg "part" is given by the other components; thus such "part" can
be calculated as:

 "part" = 1/(1+1/Lg) = 0.370

so:

 Ls' = 1*"part" = 0.370
 Ln' = (1/Lg)*"part" = 0.630 = 1.0-Ls'

 Lr' = (1.0-Ls')*Lr = 0.188
 Lg' = (1.0-Ls')*Lg = 0.370
 Lb' = (1.0-Ls')*Lb = 0.072

To calculate the component CN we can think that its contribution to Lrgb()
is equal to the contribution of CN' and *potentially* SN' to Lrgbs().
That "potentially" means that SN' has effect only when Ccol[SN']=Ccol[CN']
or every CN' if Ccol[SN'] is black.

 Lrgb(R,0,0) = Lrgbs(R',0,0,S')   ->   Lr*R = Lr'*R' [+Ls'*255*Lr]
 Lrgb(0,G,0) = Lrgbs(0,G',0,S')   ->   Lg*G = Lg'*G' [+Ls'*255*Lg]
 Lrgb(0,0,B) = Lrgbs(0,0,B',S')   ->   Lb*B = Lb'*B' [+Ls'*255*Lb]

From the formulae above we deduct the general equation:

          Lc'*C' [+Ls'*255*Lc]        Lc'*C'
 C   =    -------------------    =    ------ [+Ls'*255]
                    Lc                  Lc

which in functional notation looks like:

                 Lc'*C'
 C(C',S')   =    ------ [+Ls'*255]   =   (1.0-Ls')*C' [+Ls'*255]
```

```
                  Lc

 where the [operand] is used only if Ccol[SN']=Ccol[CN']; this also
 implies that when Ccol[SN']=black the [operand] is omitted and so,
 being Lc'<Lc, the component CN' is always darkened

To solve the PAL[] problem we just need the C(C',S') formula and the
function
               GetIntensity()
                to build a simple algorithm:

 for V=0 to 255

  <get SV' from V>             ;used by C()

  R' = GetIntensity(V,CCID[R])
  PAL[V].R = R(R',S')

  G' = GetIntensity(V,CCID[G])
  PAL[V].G = G(G',S')

  B' = GetIntensity(V,CCID[B])
  PAL[V].B = B(B',S')

 next V



 - using a value of 0.37 for Ls' gives 256 unique colors
```

1.20 3.3.11 RGBP Color Composition

```
               3.3.11   RGBP Color Composition



This format is very similar to
               RGBS
               , the difference is that instead
of "strenghtening" just a single component, we re-inforce a Pair:

Here is the specific RGBP bits allocation:

 bit #      7  6  5  4  3  2  1  0
 bit name   R1 G1 B1 P1 R0 G0 B0 P0

Let's focus on the Pns, the Pair bits:

  PV

 P1 P0   Ccol[PN]

 0  0    black
 0  1    full yellow (full red   + full green)
```

```
 1  0    full cyan   (full green + full blue)
 1  1    full purple (full blue  + full red)

Note that the component PN's P is not an "intensity" but a real color with
the non-zero RGB components at 255.

"Pure" white cannot be obtained as the brightest color is:

 R=   255   ->   RV=3   ->   R1=1, R0=1   >
 G=   255   ->   GV=3   ->   G1=1, G0=1   > %11101111
 B=   255   ->   BV=3   ->   B1=1, B0=1   >
 P=yellow   ->   PV=1   ->   P1=0, P0=1   >
```

1.21 3.3.12 RGBP Palette Settings

```
                3.3.12   RGBP Palette Settings



(the color settings listed here don't include the ones already specified
in
               section 3.3.1.3
               )

From the definition given in the
               previous section
                follows that the
arrangements for the palette are:

 COLORxx    SlcPlns        +---P---+
 -$dff180   values    PV   R   G   B

 12         %11       00     0   0   0   (black)
 13         %11       01   255 255   0   (full yellow)
 14         %11       10     0 255 255   (full cyan)
 15         %11       11   255   0 255   (full purple)

This settings cause weird "average" pixels, so we're going to study deeply
what happens.
We have two pixels (R1G1B1P1R0G0B0P0 and r1g1b1p1r0g0b0p0) attached:

 plane #   plane name   value

 2         VdoPln0                  R1 G1 B1 P1 R0 G0 B0 P0
 1         VdoPln0      R1 G1 B1 P1 R0 G0 B0 P0 r1 g1 b1 p1 r0 g0 b0 p0
                                                ^^^^^^^^^^^
                                                    avg  ^^
                                                         PV'

The Ps and ps that generate all the 4 possible PV's are listed in this
table:

 PV'  P0 p1   P       p

 0    0  0    black   black
```

```
              cyan    yellow
 1    0  1    black   cyan
              cyan    purple
 2    1  0    yellow  black
              purple  yellow
 3    1  1    yellow  cyan
              purple  purple
```

But, since P and p can be mixed in any combination (inside each PV' sub-class), we have the following table:

```
      combination
 PV'      P-p          ideal RGB average

 0    black-black        0   0   0
      black-yellow     128 128   0
      cyan-black         0 128 128
      cyan-yellow      128 255 128

 1    black-cyan         0 128 128
      black-purple     128   0 128
      cyan-cyan          0 255 255
      cyan-purple      128 128 255

 2    yellow-black     128 128   0
      yellow-yellow    255 255   0
      purple-black     128   0 128
      purple-yellow    255 128 128

 3    yellow-cyan      128 255 128
      yellow-purple    255 128 128
      purple-cyan      128 128 255
      purple-purple    255   0 255
```

It seems sensible to assign to P' the color calculated as the average of the ideal averages (approximate/idealized somewhere...):

```
 PV'  RGB average

 0      0   0   0 +
      128 128   0 +
        0 128 128 +
      128 255 128 = 256 512 256   -> [/4] ->    64 128  64

 1      0 128 128 +
      128   0 128 +
        0 255 255 +
      128 128 255 = 256 512 768   -> [/4] ->    64 128 192

 2    128 128   0 +
      255 255   0 +
      128   0 128 +
      255 128 128 = 768 512 256   -> [/4] ->   192 128  64

 3    128 255 128 +
      255 128 128 +
      128 128 255 +
```

```
      255   0 255 = 768 512 768   -> [/4] ->   192 128 192

The values just found are inserted in the palette table below:

            MskPln_          +---P'--+
 COLORxx    SlcPlns
 -$dff180   values    PV'    R   G   B

 28         %1_11     00      64 128  64
 29         %1_11     01      64 128 192
 30         %1_11     10     192 128  64
 31         %1_11     11     192 128 192
```

1.22 3.3.13 RGBP <-> RGB Conversion

```
                3.3.13   RGBP <-> RGB Conversion


Let's begin from where we left in the
               general part
               :

 Lrgb(R,G,B) = Lrgbx(R',G',B',X')

which in our case can be instanced as:

 Lrgb(R,G,B) = Lrgbp(R',G',B',P')

Then, we must choose a seemingly acceptable value for Lp' and calculate
the rest of the Lc's; we must consider that the brightest color of PN is
yellow, so the contribution of this component to the real brightness is
(Lr+Lg)*0.25 (this introduces a little error when PN is not yellow); we
know also that the remaning components, together, affect the real bright-
ness by 1/4: this means that they have 0.25/((Lr+Lg)*0.25) = 1/(Lr+Lg)
times more influence than PN, i.e. 1 "part" of the RGBx brightness is gi-
ven by PN, and the other 1/(Lr+Lg) "part" is given by the other compo-
nents; thus such "part" can be calculated as:

 "part" = 1/(1+1/(Lr+Lg)) = 0.470

so:

 Lp' = 1*"part" = 0.470
 Ln' = (1/(Lr+Lg))*"part" = 0.530 = 1.0-Lp'

 Lr' = (1.0-Lp')*Lr = 0.159
 Lg' = (1.0-Lp')*Lg = 0.311
 Lb' = (1.0-Lp')*Lb = 0.060

To calculate the component CN we can think that its contribution to Lrgb()
is equal to the contribution of CN' and *potentially* PN' to Lrgbp().
Let find out how PN' affects CN':

                   affected components
```

```
 PV'   Ccol[PN']   CN'   CCID[CN']

 0     black       all   all             (affected "negatively")
 1     yellow      RN    0
                   GN    1
 2     cyan        GN    1
                   BN    2
 3     purple      RN    0
                   BN    2
```

from which we deduct:

```
 PV'>0 and (PV'=CCID[CN'] or PV'=CCID[CN']+1 or (PV'=3 and CCID[CN']=0))
```

Examining the components separately:

```
 Lrgb(R,0,0) = Lrgbp(R',0,0,P')   ->   Lr*R = Lr'*R' [+Lp'*255*Lr]
 Lrgb(0,G,0) = Lrgbp(0,G',0,P')   ->   Lg*G = Lg'*G' [+Lp'*255*Lg]
 Lrgb(0,0,B) = Lrgbp(0,0,B',P')   ->   Lb*B = Lb'*B' [+Lp'*255*Lb]
```

From the formulae above we deduct the general equation:

$$C = \frac{Lc'*C' \ [+Lp'*255*Lc]}{Lc} = \frac{Lc'*C'}{Lc} \ [+Lp'*255]$$

which in functional notation looks like:

$$C(C',P') = \frac{Lc'*C'}{Lc} \ [+Lp'*255] = (1.0-Lp')*C' \ [+Lp'*255]$$

```
 [operand] used only when the conditional expression above is true
```

To solve the PAL[] problem we just need the C(C',P') formula and the function

```
               GetIntensity()
```

to build a simple algorithm:

```
 for V=0 to 255

  <get PV' from V>              ;used by C()

  R' = GetIntensity(V,CCID[R])
  PAL[V].R = R(R',P')

  G' = GetIntensity(V,CCID[G])
  PAL[V].G = G(G',P')

  B' = GetIntensity(V,CCID[B])
  PAL[V].B = B(B',P')

 next V
```

- using a value of 0.47 for Lp' gives 256 unique colors

1.23 3.3.14 RGB332 Color Composition

3.3.14 RGB332 Color Composition

The concept behind is: since the human eye is about two times more sensitive to the differences in red's and green's intensities than blue's, why don't we extend the range of possible red/green shades? This can be easily achieved by mapping those components on 3 bits instead of the usual 2. This method is quite different from all the previous ones, so don't be surprised by the differences you'll find here (yet, it proves to be the easiest to deal with).

Here's the specific RGB332 bits allocation:

```
bit #      7  6  5  4  3  2  1  0
bit name   R2 G2 B1 R0 R1 G1 B0 G0
```

don't be fooled by this apparently messed definition; keeping in mind that:

- red bits are R2, R1, R0
- green bits are G2, G1, G0
- blue bits are B1, B0

it's easy to see that it's not so different from the "normal" RGBx allocation: R1 G1 B1 X1 R0 G0 B0 X0.
In fact, the Xs bits have been used for the least significant bits of red and green, while the others remain the same, apart from the fact that the most significant bits of red and green have been renamed.

Let's see how the least significant bits (we'll use the letter "L" to mark them in all the RGB332 sections) must be handled:

LV

R0	G0	Ccol[LN]
0	0	some gray
0	1	some red
1	0	some green
1	1	some yellow (some red + some green)

Note that since the component BN has a "grain" different from the others, it's impossible to have perfect gray shades (white included; the brightest color, though, is still given by %11111111).

- very special thanks go to Victor Haaz for suggesting this RGBx mode and

actively contributing to its realization

1.24 3.3.15 RGB332 Palette Settings

3.3.15 RGB332 Palette Settings

Given the fact that RGB332 is a quite "special" RGBx format, its color settings don't follow the general rules specified in section 3.3.1.3 .

First of all, we must assign a normal 8-bit value to each component, because here we have 8 possible values for R and G (blue remains at 4). Let's consider the fact that when all 3 bits are ON, the value must be 255 and that when all bits are OFF the value must be 0; moreover, we can say that, for example, R2 has more influence than R1, which, on its turn, has more influence of R0: supposing that R2's weight is four times as much as R0's, and that R1's is twice as much as R0's, we can write that:

R0 + R1 + R2 = 255 -> R0 + 2*R0 + 4*R0 = 255 -> 7*R0 = 255 ->

-> R0=36.42, R1=72.85, R2=145.71

since we can only use integers, we round them to: R0=36, R1=73, R2=146

The same goes for green, while for blue the old settings apply:

B1=170, B0=85

Thus our palette settings are:

COLORxx -$dff180	SlcPlns value	CV	R	G	B	
0	%00	%00	0	0	0	
1	%00	%01	73	0	0	
2	%00	%10	146	0	0	
3	%00	%11	219	0	0	(219=73+146)
4	%01	%00	0	0	0	
5	%01	%01	0	73	0	
6	%01	%10	0	146	0	
7	%01	%11	0	219	0	(219=73+146)
8	%10	%00	0	0	0	
9	%10	%01	0	0	85	
10	%10	%10	0	0	170	
11	%10	%11	0	0	255	(255=85+170)
12	%11	%00	0	0	0	
13	%11	%01	0	36	0	
14	%11	%10	36	0	0	
15	%11	%11	36	36	0	

We now have also to recalculate the values for the colors 16-31, in case the MskPln is ON:

```
 plane #   plane name   value

 2         VdoPln0                  R2 G2 B1 R0 R1 G1 B0 G0
 1         VdoPln0      R2 G2 B1 R0 R1 G1 B0 G0 r2 g2 b1 r0 r1 g1 b0 g0
                                                ^^^^^^^^^^^
                                                ^^  avg
                                                LV'
```

For the components RN and GN, we list the intensities Cs and cs of the CVs and cvs (respectively) that generate all the 4 possible CV's:

```
  CV'

 C0 c1   C     c           how C and c have been found

 0  0      0     0         C0=0 -> CV=(%00 or %10) -> C=(  0 or 146)
         146    73         c1=0 -> cv=(%00 or %01) -> c=(  0 or  73)

 0  1      0   146         C0=0 -> CV=(%00 or %10) -> C=(  0 or 146)
         146   219         c1=1 -> cv=(%10 or %11) -> c=(146 or 217)

 1  0     73     0         C0=1 -> CV=(%01 or %11) -> C=( 73 or 217)
         219    73         c1=0 -> cv=(%00 or %01) -> c=(  0 or  73)

 1  1     73   146         C0=1 -> CV=(%01 or %11) -> C=( 73 or 217)
         219   219         c1=0 -> cv=(%10 or %11) -> c=(146 or 217)
```

But, since C and c can be mixed in any combination (inside each CV' sub-class), we have the following table:

```
      combination
 CV'   C  -  c      ideal RGB average ( (C+c)/2 )

 0      0 -   0       0
        0 -  73      37
      146 -   0      73
      146 -  73     110

 1      0 - 146      73
        0 - 219     110
      146 - 146     146
      146 - 219     183

 2     73 -   0      37
       73 -  73      73
      219 -   0     110
      219 -  73     146

 3     73 - 146     110
       73 - 219     146
      219 - 146     183
      219 - 219     219
```

It seems sensible to assign to C' the intensity calculated as the average
of its ideal averages (approximate/idealized somewhere...):

```
 CV'  C'

 0      0 +
       36 +
       73 +
      110 = 219   -> [/4] ->    55

 1     73 +
      110 +
      146 +
      183 = 512   -> [/4] ->   128

 2     36 +
       73 +
      110 +
      146 = 365   -> [/4] ->    91

 3    110 +
      146 +
      183 +
      219 = 658   -> [/4] ->   165
```

The palette settings for red and green are:

COLORxx -$dff180	MskPln_ SlcPlns value	CV	R	G	B
16	%1_00	%00	55	0	0
17	%1_00	%01	128	0	0
18	%1_00	%10	91	0	0
19	%1_00	%11	165	0	0
20	%1_01	%00	0	55	0
21	%1_01	%01	0	128	0
22	%1_01	%10	0	91	0
23	%1_01	%11	0	165	0

For blue, we'll use the values in
section 3.3.1.3
(64, 149, 107, 192):

COLORxx -$dff180	MskPln_ SlcPlns value	CV	R	G	B
24	%1_10	%00	0	0	64
25	%1_10	%01	0	0	149
26	%1_10	%10	0	0	107
27	%1_10	%11	0	0	192

Instead some more calculations are needed to cope with the "average" pixels
caused by the R0 and G0 bits:

```
 plane #   plane name   value

 2         VdoPln0                  R2 G2 B1 R0 R1 G1 B0 G0
 1         VdoPln0      R2 G2 B1 R0 R1 G1 B0 G0 r2 g2 b1 r0 r1 g1 b0 g0
                                                ^^^^^^^^^^
                                                    avg  ^^
                                                         LV'
```

The G0 and r0 that generate all the 4 possible LV's are listed in this table:

LV'	G0	r0	L	l
0	0	0	black	black
			red	green
1	0	1	black	red
			red	yellow
2	1	0	green	black
			yellow	green
3	1	1	green	red
			yellow	yellow

But, since L and l can be mixed in any combination (inside each LV' sub-class), we have the following table:

LV'	combination L-l	ideal RGB average		
0	black-black	0	0	0
	black-green	0	18	0
	red-black	18	0	0
	red-green	18	18	0
1	black-red	18	0	0
	black-yellow	18	18	0
	red-red	36	0	0
	red-yellow	36	18	0
2	green-black	0	18	0
	green-green	0	36	0
	yellow-black	18	18	0
	yellow-green	18	36	0
3	green-red	18	18	0
	green-yellow	18	36	0
	yellow-red	36	18	0
	yellow-yellow	36	36	0

It seems sensible to assign to L' the color calculated as the average of the ideal averages (approximate/idealized somewhere...):

```
 LV'  RGB average

 0      0   0   0 +
        0  18   0 +
       18   0   0 +
       18  18   0 =  36  36   0   -> [/4] ->     9   9   0
```

```
 1      18   0   0 +
        18  18   0 +
        36   0   0 +
        36  18   0 = 108  36   0   -> [/4] ->    27   9   0

 2       0  18   0 +
         0  36   0 +
        18  18   0 +
        18  36   0 =  36 108   0   -> [/4] ->     9  27   0

 3      18  18   0 +
        18  36   0 +
        36  18   0 +
        36  36   0 = 108 108   0   -> [/4] ->    27  27   0
```

The remaining part of the palette becomes:

```
            MskPln_
 COLORxx    SlcPlns
 -$dff180   value      CV   R   G   B

 28         %1_11     %00    9   9   0
 29         %1_11     %01   27   9   0
 30         %1_11     %10    9  27   0
 31         %1_11     %11   27  27   0
```

I think that a good summary could be useful:

```
            MskPln_
 COLORxx    SlcPlns
 -$dff180   value      CV   R   G   B

 0          %00       %00     0   0   0
 1          %00       %01    73   0   0
 2          %00       %10   146   0   0
 3          %00       %11   219   0   0   (219=73+146)

 4          %01       %00     0   0   0
 5          %01       %01     0  73   0
 6          %01       %10     0 146   0
 7          %01       %11     0 219   0   (219=73+146)

 8          %10       %00     0   0   0
 9          %10       %01     0   0  85
 10         %10       %10     0   0 170
 11         %10       %11     0   0 255   (255=85+170)

 12         %11       %00     0   0   0
 13         %11       %01     0  36   0
 14         %11       %10    36   0   0
 15         %11       %11    36  36   0

 16         %1_00     %00    55   0   0
 17         %1_00     %01   128   0   0
 18         %1_00     %10    91   0   0
 19         %1_00     %11   165   0   0
```

```
20          %1_01      %00       0   55    0
21          %1_01      %01       0  128    0
22          %1_01      %10       0   91    0
23          %1_01      %11       0  165    0

24          %1_10      %00       0    0   64
25          %1_10      %01       0    0  149
26          %1_10      %10       0    0  107
27          %1_10      %11       0    0  192

28          %1_11      %00       9    9    0
29          %1_11      %01      27    9    0
30          %1_11      %10       9   27    0
31          %1_11      %11      27   27    0
```

- a significant drawback of this method is a noticeable loss of brightness, due to the fact that a) we use much less than the max available brightness of each pixel (in fact settings are very low); b) RN and BN components actually add up using two different pixels quite distant from each other: maybe a different arrangement of bits could give better results...
- on the other side, this method offers an almost perfect balancement of colors (no component is "overshadowed" by the other components) and very smooth graduation of RN and GN
- 256 unique colors

1.25 3.3.16 RGB332 <-> RGB Conversion

3.3.16 RGB332 <-> RGB Conversion

Given that this method uses only red, green and blue, the RGB <-> RGB332 conversion is much easier than the other format's; everything comes to finding the corrispondence between the same components in the two formats.

We'll proceed following the same reasoning described here.

In RGB there are 256 possible values per component, while in RGB332 there are only 8 for red and green and just 4 for blue. Considering these two cases separately:

```
R/G = {0, 36, 73, 109, 146, 182, 219, 255}
B   = {0, 85, 170, 255}
```

Intuitively we can say that if a component has a certain value in RGB then the corresponding value in RGB332 must be the the closest among those listed above. For example, let's suppose that red's intensity in RGB is 123; the closest value available in RGB332 is 109. The same value for blue would be converted to 85. In general, we have to pick the RGB332 intensity

that contains the RGB intensity in its "surroundings". These intervals can be defined by diving [0...255] in this way:

```
 RGB        RGB332   RGB332       RGB        RGB332   RGB332
 R/G         R/G     RV/GV         B           B        BV

   0 --+-->   0      0             0 --+-->   0        0
  18 --+                           42 --+
       +-->  36      1                 +-->  85        1
  54 --+                           127 --+
       +-->  73      2                 +--> 170        2
  90 --+                           212 --+
       +--> 109      3             255 --+--> 255      3
 126 --+
       +--> 146      4
 162 --+
       +--> 182      5
 198 --+
       +--> 219      6
 234 --+
 255 --+--> 255      7
```

from which we can derive these formulas:

```
 RV3 = (R8+17)/36   <->   R8 = 36*RV3
 GV3 = (G8+17)/36   <->   G8 = 36*GV3
 BV2 = (B8+42)/85   <->   B8 = 85*BV2      (the same of
               C2CV()
               )
```

(in all these tables and formulas there are some errors due to the fact that 255 isn't multiple of neither 36 nor 85)

Supposing to have the function GetCV(x,CN) which returns the CV of the component CN of the RGB332 value x, the algo to build the PAL[] table is simply:

```
 for V=0 to 255

  PAL[V].R = 36*GetCV(V,RN)
  PAL[V].G = 36*GetCV(V,GN)
  PAL[V].B = 85*GetCV(V,BN)

 next V
```

1.26 3.3.17 RGBH Color Composition

3.3.17 RGBH Color Composition

This mode is a sort of "hacked RGB332"; we have seen, in fact, that the bad side of that mode is the lack of brightness: so, why don't we "boost" it?

```
Obviously in this way we'll lose the perfect balancement of components which
caracterizes the
               RGB332 mode
               , but generally a bright, although "uncorrect",
method is preferable to an "exact" but too dark one. That's why I decided
to pull out this "ultra" version.
Luckily, we already know how to do all our calculations: we'll just need
Uns bits to "strenghten" the RN and GN components in a way analougous to

               RGBM
               .

Here is the specific RGBH bits allocation:

 bit #      7  6  5  4  3  2  1  0
 bit name   R1 G1 B1 U1 R0 G0 B0 H1

Let's focus on the Hns, the Hack bits:

  HV

 H1 H0   Ccol[HN]

 0  0    black
 0  1    full red
 1  0    full green
 1  1    full white

By doing this, the most influencing components (RN and GN) get additional
"charge" by the component HN, whose H ("irregularly") is not a simple in-
tensity, but a real color with the non-zero RGB components always at 255.

In this way, a strong red would be:

 R=255   ->   RV=3   ->   R1=1, R0=1   >
 G=  0   ->   GV=0   ->   G1=0, G0=0   > %10001001
 B=  0   ->   BV=0   ->   B1=0, B0=0   >
 H=red   ->   HV=1   ->   H1=0, H0=1   >

Equally, a strong green is given by:

 R=    0   ->   RV=0   ->   R1=0, R0=0   >
 G=  255   ->   GV=3   ->   G1=3, G0=0   > %01010101
 B=    0   ->   BV=0   ->   B1=0, B0=1   >
 H=green   ->   HV=2   ->   H1=1, H0=0   >

White, like in most modes, is given by %11111111:

 R=  255   ->   RV=3   ->   R1=1, R0=1   >
 G=  255   ->   GV=3   ->   G1=1, G0=1   > %11111111
 B=  255   ->   BV=3   ->   B1=1, B0=1   >
 H=white   ->   HV=3   ->   H1=1, H0=1   >
```

1.27 3.3.18 RGBH Palette Settings

3.3.18 RGBH Palette Settings

(the color settings listed here don't include the ones already specified in section 3.3.1.3)

From the definition given in the previous section follows that the arrangements for the palette are:

COLORxx -$dff180	SlcPlns values	UV	+---H---+ R	G	B	
12	%11	00	0	0	0	(black)
13	%11	01	255	0	0	(full red)
14	%11	10	0	255	0	(full green)
15	%11	11	255	255	255	(full white)

This settings cause weird "average" pixels, so we're going to study deeply what happens.
We have two pixels (R1G1B1H1R0G0B0H0 and r1g1b1h1r0g0b0h0) attached:

plane #	plane name	value
2	VdoPln0	R1 G1 B1 H1 R0 G0 B0 H0
1	VdoPln0	R1 G1 B1 H1 R0 G0 B0 H0 r1 g1 b1 h1 r0 g0 b0 h0

```
                                          ^^^^^^^^^^^
                                              avg  ^^
                                                   HV'
```

The Hs and hs that generate all the 4 possible HV's are listed in this table:

HV'	H0	h1	H	h
0	0	0	black	black
			green	red
1	0	1	black	green
			green	white
2	1	0	red	black
			white	red
3	1	1	red	green
			white	white

But, since H and h can be mixed in any combination (inside each HV' sub-class), we have the following table:

HV'	combination HV-hv	ideal RGB average		
0	black-black	0	0	0
	black-red	128	0	0

```
      green-black     0 128   0
      green-red     128 128   0

 1    black-green     0 128   0
      black-white   128 128 128
      green-green     0 255   0
      green-white   128 255 128

 2    red-black     128   0   0
      red-red       255   0   0
      white-black   128 128 128
      white-red     255 128 128

 3    red-green     128 128   0
      red-white     255 128 128
      white-green   128 255 128
      white-white   255 255 255
```

It seems sensible to assign to H' the color calculated as the average of the ideal averages (approximate/idealized somewhere...):

```
 HV'  RGB average

 0      0   0   0 +
      128   0   0 +
        0 128   0 +
      128 128   0 = 256 256 000   -> [/4] ->    64  64   0

 1      0 128   0 +
      128 128 128 +
        0 255   0 +
      128 255 128 = 256 768 255   -> [/4] ->    64 192  64

 2    128   0   0 +
      255   0   0 +
      128 128 128 +
      255 128 128 = 768 256 256   -> [/4] ->   192  64  64

 3    128 128   0 +
      255 128 128 +
      128 255 128 +
      255 255 255 = 768 768 512   -> [/4] ->   192 192 128
```

The values just found are inserted in the palette table below:

```
            MskPln_          +---H'--+
 COLORxx    SlcPlns
 -$dff180   values    HV'   R   G   B

 28         %1_11     00     64  64   0
 29         %1_11     01     64 192  64
 30         %1_11     10    192  64  64
 31         %1_11     11    192 192 128
```

1.28 3.3.19 RGBH <-> RGB Conversion

3.3.19 RGBH <-> RGB Conversion

Let's begin from where we left in the general part :

Lrgb(R,G,B) = Lrgbx(R',G',B',X')

which in our case can be instanced as:

Lrgb(R,G,B) = Lrgbh(R',G',B',UI'(CID))

where HI'(CID) = intensity of the component #CID in color H' = 255*Lc' (H' couldn't be used directly because in this format it is a real color; the fixed value 255 comes from the fact that H' is always a full color, except in the case of black, which will be treated appropriately later)

Then, we must choose a seemingly acceptable value for Lh' and calculate the rest of the Lc's; we must consider that the brightest color of HN is white, so, similarly to RGBW, we could set Lh' to 0.5 (this introduces a little error when HN is not white), but we'll use 0.49 to get 256 unique colors:

Lh' = 0.49
Lr' = (1.0-Lh')*Lr = 0.153
Lg' = (1.0-Lh')*Lg = 0.299
Lb' = (1.0-Lh')*Lb = 0.058

To calculate the component CN, we can think that its contribution to Lrgb() is equal to the contribution of CN' and *potentially* HN' to Lrgbh(). "Potentially" means that UN' has effect when Ccol[HN']=Ccol[CN'] or when Ccol[HN']=black or Ccol[HN']=white.

Lrgb(R,0,0) = Lrgbh(R',0,0,UI'(0)) -> Lr*R = Lr'*R' [+Lh'*255*Lr]
Lrgb(0,G,0) = Lrgbh(0,G',0,UI'(1)) -> Lg*G = Lg'*G' [+Lh'*255*Lg]
Lrgb(0,0,B) = Lrgbh(0,0,B',UI'(2)) -> Lb*B = Lb'*B' [+Lh'*255*Lb]

From the formulae above we deduct the general equation:

$$C = \frac{Lc'*C' \ [+Lh'*255*Lc]}{Lc} = \frac{Lc'*C'}{Lc} \ [+Lh'*255]$$

which in functional notation looks like:

$$C(C',HI'(CID)) = \frac{Lc'*C'}{Lc} \ [+Lh'*255] = (1.0-Lh')*C' \ [+Lh'*255]$$

where the [operand] is used only if Ccol[HN']=Ccol[CN'] or when

```
 Ccol[HN']=white; this also means that when Ccol[HN']=black the [operand]
 is omitted and so, being Lc'<Lc, the CN' is always darkened no matter
 which component it is

To solve the PAL[] problem we just need the C(C',HI(CID)') formula and the
function
               GetIntensity()
                to build a simple algorithm:

 for V=0 to 255

  <get H' and HV' from V>              ;used by HI'() and C()

  R' = GetIntensity(V,CCID[R])
  PAL[V].R = R(R',HI'(0))

  G' = GetIntensity(V,CCID[G])
  PAL[V].G = G(G',HI'(1))

  B' = GetIntensity(V,CCID[B])
  PAL[V].B = B(B',HI'(2))

 next V



 - using a value of 0.49 for Lh' gives 256 unique colors
```

1.29 3.4 Improving Picture Quality with ChqrMode

```
                3.4   Improving Picture Quality with ChqrMode



*********************************************************************
* The content of this paragraph applies to HalfRes displays only *
*********************************************************************


We have seen in
               section 3.3.1.3
                that in HalfRes mode 2 horizontally con-
secutive pixels are separated by a "special" pixel generated by the pixels
themselves.

Try to imagine a horizontal line of the screen:

 A·B·C·D·E·F·G·H·I·J·K·L·M·N·O·P·Q·R·S·T·U·V·W·X·Y·Z·

 where the capital letters represent the pixels and the '·'s their "ave-
 rage" pixels

Even though we know how to horizontally-wise
               make the best of the '·'
```

,
we should consider them also vertically:

```
A·B·C·D·E·F·G·H·I·J·K·L·M·N·O·P·Q·R·S·T·U·V·W·X·Y·Z·
S·T·U·V·W·X·Y·Z·A·B·C·D·E·F·G·H·I·J·K·L·M·N·O·P·Q·R·
S·T·U·V·W·X·H·I·J·K·L·M·N·O·P·Q·R·Y·Z·A·B·C·D·E·F·G·
Q·R·Y·Z·A·B·C·D·E·F·G·S·T·U·V·W·X·H·I·J·K·L·M·N·O·P·
B·X·H·I·J·K·L·M·N·O·Q·R·Y·Z·A·P·C·D·E·F·G·S·T·U·V·W·
Q·R·Y·Z·A·B·X·H·I·J·K·L·M·N·O·P·C·D·E·F·G·S·T·U·V·W·
F·G·S·T·U·V·W·Q·R·Y·Z·A·B·X·H·I·J·K·L·M·N·O·P·C·D·E·
R·Y·Z·A·B·X·H·I·J·K·L·M·N·O·P·C·D·E·F·G·S·T·U·V·W·Q·
J·K·L·M·N·O·P·C·D·E·F·G·S·T·U·V·W·Q·R·Y·Z·A·B·X·H·I·
N·O·P·C·D·E·F·G·S·J·K·L·M·T·U·V·W·Q·R·Y·Z·A·B·X·H·I·
F·G·S·J·K·L·M·T·U·V·W·Q·R·Y·Z·A·B·X·H·I·N·O·P·C·D·E·
L·M·T·U·V·W·Q·R·Y·Z·A·B·X·H·I·N·O·P·C·D·E·F·G·S·J·K·
E·F·G·S·J·K·L·M·T·U·V·W·Q·R·Y·Z·A·B·X·H·I·N·O·P·C·D·
V·W·Q·R·Y·Z·A·B·X·H·I·N·O·P·C·D·E·F·G·S·J·K·L·M·T·U·
B·X·H·I·N·O·P·C·D·E·F·G·S·J·K·L·M·T·U·V·W·Q·R·Y·Z·A·
E·F·G·S·J·K·L·M·T·U·V·W·Q·R·Y·Z·A·B·X·H·I·N·O·P·C·D·
```

Generally it's not a good effect having columns of pixels alternated with columns of "average" pixels.
Most of the times it's enough to shift the even (or odd - no difference) lines by 1 pixel to achive a much better looking screen:

```
A·B·C·D·E·F·G·H·I·J·K·L·M·N·O·P·Q·R·S·T·U·V·W·X·Y·Z·
 S·T·U·V·W·X·Y·Z·A·B·C·D·E·F·G·H·I·J·K·L·M·N·O·P·Q·R
S·T·U·V·W·X·H·I·J·K·L·M·N·O·P·Q·R·Y·Z·A·B·C·D·E·F·G·
 Q·R·Y·Z·A·B·C·D·E·F·G·S·T·U·V·W·X·H·I·J·K·L·M·N·O·P
B·X·H·I·J·K·L·M·N·O·Q·R·Y·Z·A·P·C·D·E·F·G·S·T·U·V·W·
 Q·R·Y·Z·A·B·X·H·I·J·K·L·M·N·O·P·C·D·E·F·G·S·T·U·V·W
F·G·S·T·U·V·W·Q·R·Y·Z·A·B·X·H·I·J·K·L·M·N·O·P·C·D·E·
 R·Y·Z·A·B·X·H·I·J·K·L·M·N·O·P·C·D·E·F·G·S·T·U·V·W·Q
J·K·L·M·N·O·P·C·D·E·F·G·S·T·U·V·W·Q·R·Y·Z·A·B·X·H·I·
 N·O·P·C·D·E·F·G·S·J·K·L·M·T·U·V·W·Q·R·Y·Z·A·B·X·H·I
F·G·S·J·K·L·M·T·U·V·W·Q·R·Y·Z·A·B·X·H·I·N·O·P·C·D·E·
 L·M·T·U·V·W·Q·R·Y·Z·A·B·X·H·I·N·O·P·C·D·E·F·G·S·J·K
E·F·G·S·J·K·L·M·T·U·V·W·Q·R·Y·Z·A·B·X·H·I·N·O·P·C·D·
 V·W·Q·R·Y·Z·A·B·X·H·I·N·O·P·C·D·E·F·G·S·J·K·L·M·T·U
B·X·H·I·N·O·P·C·D·E·F·G·S·J·K·L·M·T·U·V·W·Q·R·Y·Z·A·
 E·F·G·S·J·K·L·M·T·U·V·W·Q·R·Y·Z·A·B·X·H·I·N·O·P·C·D
```

The '·'s are much less visible and more integrated with the rest of the picture.

The Amiga with its BPLCON1 register helps a lot in this case (as in many others...): normally, at each line, it would be enough to load it alternatively with 0 and $11 to get what we want.
Yet, a TCS display already requires the BPLCON1 to be set to $10, so what can we do? Well, nothing really changes: $10 and $21 will do the job quite well.

Most importantly, instead, one should consider how to build up the copperlist which carries out that job; we have 2 possible choices:

- a Copper loop which waits the beginning of every rasterline and than COPMOVEs the right value to BPLCON1

- a simple list of couples of COPWAITs and COPMOVEs: each of them waits for its own rasterline and then writes to BPLCON1

The copper-loop method looks less stupid, but indeed it requires many more instruction fetches by the Copper and so more CHIP ram bus usage (as far as I know, Copper hasn't an instruction cache!), which is entirely to CPU disadvantage (see the test results for further information about the ChqrMode influence on performance).
Not only that, but a Copper loop would make
horizontal scrolling
impos-
sible (unless forcing the CPU to intervene (for example with a Copper interrupt) at every loop)!!!
For these reasons, the best method is surely the banal and longer list of COPMOVEs that can be easily implemented with a series of DsplHt/2 (DsplHt = display height in lines) chunks of this kind:

```
dc.w   $xxe1,$fffe     ;wait rasterline $xx end ($xx is even)
dc.w   $0102,$0010     ;write $0010 to BPLCON1

dc.w   $yye1,$fffe     ;wait rasterline $yy end ($yy = ($xx+1)and$ff)
dc.w   $0102,$0021     ;write $0021 to BPLCON1
```

- note that since BPLCON1 indicates the shift rightward, the left border looks a bit "ragged": to eliminate this little side-effect, shift by 1 or 2 LORES pixels the display's horizontal start position using DIWSTRT and DIWHIGH
- there's no difference in shifting righward of leftward: the only thing that would change is the ragged side
- this technique has a negative effect on simple/geometrical images due the bad look of (almost) vertical lines

1.30 3.5 Creating Scrollable Screens

3.5 Creating Scrollable Screens

Here we'll deal with a quite hard subject: how can we scroll a screen larger than the TCS display?

To start, we should consider whether scrolling is really important: TCS, in fact, was born with everthing but scrolling in mind. This is because I don't consider the capability scrolling a fundamental feature for screens that are probably going to be used for demo effects, 3D graphics and all this kind of stuff.

Anyway, Amiga has always been reknown for its smooth scroll and the ease of producing such effect: we can't absolutely forget this issue.

3.5.1
Scrolling in FullRes

1.31 3.5.1 Scrolling in FullRes

3.5.1 Scrolling in FullRes

If we look at the
setup
needed to open a FullRes display, we notice that
there is no particular setting so we could scroll as we normally would with a "classic" Amiga screen (BPLCON1 can be used without restrictions of sort and vertical scrolling can be achieved as usual).
Yet, as we'll see, this would waste a large amount of CHIP mem; instead, there is another way which (almost) for free lets us scroll easily and, at the same time, saving a lot of memory: recalling what has been discussed in the
FullRes specific section
, we can think of taking advantage of the
conversion routine itself: it is enough, in fact, to select a different area to convert from time to time to produce the desired effect. The only negative side of this method is that the area selected for conversion lies often (about 3/4 of the times) on a horizontal boundary not longword aligned, thus causing a little speed loss if source pixels are fetched by groups of 4 (the 68k has to do 2 memory reads for misaligned accesses).

1.32 3.5.2 Vertical Scrolling in HalfRes

3.5.2 Vertical Scrolling in HalfRes

As for vertical scrolling, nothing is really new: it's enough to change the values stored in the BPLxPT registers by adding PlnWd*YPos to the planes addresses, where PlnWd is the width in bytes of a plane and YPos the first line to be shown at the top of the display.

1.33 3.5.3 Horizontal Scrolling in HalfRes

3.5.3 Horizontal Scrolling in HalfRes

Unfortunately things start to take a bad shape when it comes to horizontal scrolling. Why? Because of all the specific settings needed by HalfRes displays; in particular, what makes everything complicated are the play-

```
fields' different shift values (they must be shifted of 1 LORES pixel in
respect to each other,
               remember
               ?) and the
               chequer effect
                (it requires a

               long copperlist
               ).

Playfields shifting is compulsory, so the cases to study are just two.
One thing these two cases have in common is that 8 extra bytes have to be
early-fetched before the display window start: this means that the DDFSTRT
value must be lower than in the no-scroll case, that the BPLxMOD registers
must be loaded with a negative value to "rescue" the extra bytes fetched
and that the BPLxPTs must point 8 bytes before the actual start of the
planes.

3.5.3.1
                Scrolling with ChqrMode OFF
                3.5.3.2
                Scrolling with ChqrMode ON


 - reading 8 bytes more per line (for each bitplane) takes longer CHIP ram
   bus time
```

1.34 3.5.3.1 Scrolling with ChqrMode OFF

```
                3.5.3.1   Scrolling with ChqrMode OFF


In this case the only obstacle we have to face is that playfield 2 must be
shifted, _rightward_, of 1 LORES pixel more than playfield 1. Normally
(i.e. when scrolling is not required), this simply translates to setting
BPLCON1 to $10; in our case, though, neither this nor keeping just a $10
difference between the lowest nibbles of the register is no longer enough.
Let's see why recalling briefly how to obtain a normal horizontal scroll
of XPos SHRES pixels _leftwards_:

 we have to calculate:

  - the shift value (for each playfield) to write in BPLCON1 (remembering
    that H6 and H7 don't work in SHRES, hence 0<=shift<=63 SHRES pixels)
  - the offset, expressed in bytes, from the base address of any plane,
    which will be used to determine the addresses of the planes' first
    bytes to be fetched by the DMA (these address are written to the
    BPLxPT registers)

  XPos   shift in LORES pixels   shift(XPos)   offset(XPos)

  0      00.00                   0             0
  1      15.75                   63            +8
  2      15.50                   62            +8
```

```
  3      15.25                    61            +8
  4      15.00                    60            +8
  .      .                        .             .
  .      .                        .             .
  .      .                        .             .
  63     00.25                    1             +8
  64     00.00                    0             +8
  65     15.75                    63            +16
  .      .                        .             .
  .      .                        .             .
  .      .                        .             .
  255    00.25                    1             +56
  256    00.00                    0             +56
  257    15.75                    63            +64
  .      .                        .             .
  .      .                        .             .
  .      .                        .             .
```

from this table we deduct that (the notation "(x)" means "modulo x"):

- shift(x) = (64-(x (64)) (64) = (64-(x and 63)) and 63 =
  = -(x (64)) (64) = -(x and 63) and 63
- offset(x) = ((x+63) and not 63)/8 = ((x+63) and $ffc0)>>3

and (the function "ShfCON1(z)", given the shift value 'z', returns the shift value in the format accepted by BPLCON1 for playfield 1; PlnAdr = base address of a generic bitplane) that:

- BPLCON1 = ShfCON1(shift(XPos)) or ShfCON1(shift(XPos))<<4
- BPLxPT = PlnAdr + offset(XPos) -8

As for our problem of keeping a shift difference of 1 LORES pixel, it's not so intuitive that playfield 1 has to be shifted 1 pixel left (rather than shifting playfield 2 one pixel right); then, looking at the table, we see that the first four lines illustrate why a simple $10 in BPLCON1 wouldn't be enough: not only BPLCON1 should be $000f (because the playfield to shift is changed, shift(XPos)=0 and the LORES shift for XPos+4 is 15), but also the BPLxPTs change (because offset(XPos) <> offset(XPos+4))! This means that we'll have to calculate those values separately for each playfield and then store them in the copperlist in the data fields of the appropriate COPMOVEs:

- BPLCON1 = ShfCON1(shift(XPos+4)) or ShfCON1(shift(XPos))<<4
- BPLxPT = PlnAdr + offset(XPos+4) -8 (playfield 1)
  BPLxPT = PlnAdr + offset(XPos) -8 (playfield 2)

In the next section we'll need them in functional notation:

- GetBPLCON1(x) = ShfCON1(shift(x+4)) or ShfCON1(shift(x))<<4
- GetBPLxPT(x) = PlnAdr + offset(x) -8

- note that the offsets added to PlnAdr (possibly inclusive of the verti-

cal offset) should be kept in a safe place in case other operations like planes swap for double/triple buffering, etc. must be done without re-executing the calculations above (with a single copperlist)

1.35 3.5.3.2 Scrolling with ChqrMode ON

3.5.3.2 Scrolling with ChqrMode ON

Well, after all, the
ChqrMode OFF case
was not so complicated, so we can
be quite optimistic, right?
NOT!!!

Our target is still scrolling the screen by XPos SHRES pixels left.
Let's start keeping in mind that all that's been said in
that section
still holds; then, let's find out what changes in the piece of copperlist
shown
here
when the scrolling is enabled:

```
     dc.w   $xxe1,$fffe     ;wait rasterline $xx end ($xx is even)
     dc.w   $0102,$ssss     ;write $ssss to BPLCON1

     dc.w   $yye1,$fffe     ;wait rasterline $yy end ($yy = ($xx+1) (256))
     dc.w   $0102,$tttt     ;write $tttt to BPLCON1
```

It's clear that to scroll horizontally and contemporarily having the even lines shifted of 1 LORES pixel (leftward, this time) more than the odd ones, the 68k has to write to each and every COPMOVE in those chunks (for a total of DsplHt writes)
GetBPLCON1(XPos)
(in the place of $ssss) and

GetBPLCON1(XPos+4)
(in the place of $tttt).

But we know that this wouldn't be enough: we should also update the BPLxPT registers! Immeditalely one starts thinking to add some more COPMOVEs to reload such registers, but that would result in an incredible amount of Copper istructions, since at least 4 (5 if MskPln is active) BPLxPTs need to be updated, for a total of 4*2*2 (5*2*2) additional COPMOVEs per chunk! More job for both the Copper and the 68k!
Luckily we can do better. A much more feasible solution would be adding a couple of COPMOVEs to the BPLxMOD registers per scanline (so we add only 2*2 COPMOVEs per chunk):

```
     dc.w   $xxe1,$fffe     ;wait rasterline $xx end ($xx is even)
     dc.w   $0102,$ssss     ;write $ssss=
```
GetBPLCON1(XPos)
to BPLCON1

```
     dc.w   $0108,$mmmm     ;write $mmmm to BPL1MOD
     dc.w   $010a,$nnnn     ;write $nnnn to BPL2MOD

     dc.w   $yye1,$fffe     ;wait rasterline $yy end ($yy = ($xx+1) (256))
     dc.w   $0102,$tttt     ;write $tttt=
               GetBPLCON1(XPos+4)
                to BPLCON1
     dc.w   $0108,$pppp     ;write $pppp to BPL1MOD
     dc.w   $010a,$qqqq     ;write $qqqq to BPL2MOD
```

what are they supposed to do, now? Let’s put them aside for a moment.

Instead, for a second we have to reconsider the way we used
```
               GetBPLCON1()
               .
```
We applied it to both XPos and XPos+4: we must pay attention because the function itself internally adds 4 to its input value when calling
```
               shift()
                to calculate the shift of playfield1. Look at what happens:

 line   x        GetBPLCON1(x)

 odd    XPos     ShfCON1(
               shift(XPos+4)
               ) or ShfCON1(
               shift(XPos)
               )<<4
 even   XPos+4   ShfCON1(
               shift(XPos+8)
               ) or ShfCON1(
               shift(XPos+4)
               )<<4
```

If fact when the line is odd, playfield 1 must be shifted of 4 pixels; when the line is even, playfield 2 must be shifted of 4 pixels and playfield 1 of 8 pixels:

```
 line   playfield   shift   depends on

 odd    1            0
 odd    2           +4
               how pixels are formed
                 even   1           +4      chequer
 even   2           +8
               how pixels are formed
                + chequer


               shift()
                contains a modulo operation inside because when the input value
```
reaches or goes beyond 64, the scrolling is obtained with the help of the bitplanes pointers, which we were discussing above. This observation is useful to find out when the offsets are equal in both the even and odd lines; so, having a look at the
```
               table of the shifts and offsets
               , we have
```
to find the possible values for XPos that don’t produce a planes pointers

```
change (that in maths means: XPos (64) +4 < 65 -> XPos (64) < 61 and
XPos (64) +8 < 65 -> XPos (64) < 57):

 -   1 <= XPos (64) <= 56:

      lines   playfield   offset

      odd     2
               offset(XPos)
                      odd     1
               offset(XPos+4)
                =
               offset(XPos)
                      even    2
               offset(XPos+4)
                =
               offset(XPos)
                      even    1
               offset(XPos+8)
                =
               offset(XPos)
                 -   57 <= XPos (64) <= 60:

      lines   playfield   offset

      odd     2
               offset(XPos)
                      odd     1
               offset(XPos+4)
                =
               offset(XPos)
                      even    2
               offset(XPos+4)
                =
               offset(XPos)
                      even    1
               offset(XPos+8)
                =
               offset(XPos)
               +8

 -   61 <= XPos (64) <= 64 = 0:

      lines   playfield   offset

      odd     2
               offset(XPos)
                      odd     1
               offset(XPos+4)
                =
               offset(XPos)
               +8
      even    2
               offset(XPos+4)
                =
               offset(XPos)
               +8
```

```
      even    1
               offset(XPos+8)
                =
               offset(XPos)
               +8
```

in human words, we have found that the BPLxPTs are affected in 3 different ways:

- if XPos (64) belongs to {1,...,56} then both the playfieds have the same offset, no matter whether the line number is odd or even
- if XPos (64) belongs to {57,...,60} then the playfields share the same offset only on the odd lines
- if XPos (64) belongs to {0,61,62,63} then the playfields share the same offset only on the even lines, and this offset is greater than the one needed by XPos

Now we're getting closer to the solution: the BPLxPT registers can be loaded just once with the values given by

```
               GetBPLxPT()
                for every
```

bitplanes, before the series of the copperlist chunks and the BPLxMOD registers can be used to add the +8 bytes difference when needed.
Supposing to start from an odd line, the BPLxPTs settings are:

```
 playfield   XPos (64) in   BPLxPTs

 2           {1,...,56}
               GetBPLxPT(XPos)
                 1           {1,...,56}
               GetBPLxPT(XPos+4)
                =
               GetBPLxPT(XPos)
                 2           {57,...,60}
               GetBPLxPT(XPos)
                 1           {57,...,60}
               GetBPLxPT(XPos+4)
                =
               GetBPLxPT(XPos)
                 2           {0,61,62,63}
               GetBPLxPT(XPos)
                 1           {0,61,62,63}
               GetBPLxPT(XPos+4)
                =
               GetBPLxPT(XPos)
               +8
```

The only things that remain to discover are the values to assign to the BPLxMOD registers (let's remember that we're fetching 8 extra-bytes, so the base value for BPLxMOD will be -8):

```
3.5.3.2.1
                XPos (64) Belongs to {1,...,56}
                3.5.3.2.2
                XPos (64) Belongs to {57,...,60}
                3.5.3.2.3
                XPos (64) Belongs to {0,61,62,63}
```

```
                Don't worry if all this mess looks obscure: the section below  ↩
                   directly
reveals the necessary settings without any explanation:

3.5.3.2.4
                Settings Summary


 - since enabling the scrolling enlarges the copperlist size, the Copper's
   accesses to the CHIP ram bus increase, too, with the consequent slow-
   down of CPU accesses to the same kind of memory
 - there is no particular restriction of the choice between odd and even
   lines: the one adopted here is just a convention
```

1.36 3.5.3.2.1 XPos (64) Belongs to {1,...,56}

```
                3.5.3.2.1   XPos (64) Belongs to {1,...,56}



               This table
                perfectly shows that in this case the offsets are the same for
any line and both the playfields:
               $mmmm = $nnnn = $pppp = $qqqq
                = -8 =
= BPL1MOD = BPL2MOD carry out the job quite well as the DMA fetches 8
bytes more than the actual width of a line.
```

1.37 3.5.3.2.2 XPos (64) Belongs to {57,...,60}

```
                3.5.3.2.2   XPos (64) Belongs to {57,...,60}



               In this case
                the playfield 2 has the same offset independently from the
line: by setting its to
               offset(XPos)
               , it's enough to have a modulo of -8
to "recover" the "extra-fetched" bytes on both the even and odd lines.

To find the right values for  BPL1MOD, we follow the process of DMA data
fetching step-by-step:

 starting from odd line;
 base offset: ofs =
               offset(XPos)
                 playfield 1 offset: ofs1 =
               offset(XPos+4)
                =
```

```
               offset(XPos)
                = ofs;

 After fetching a line, the BPLxPTs of playfield 1 point to the start
 of the next line plus ofs1 plus 8 "extra-fetched" bytes, i.e. the start
 of the next line plus ofs+8; since the next line is even, the playfield
 must have an offset equal to ofs+8: we have already reached such figure
 so a modulo of 0 is simply what is needed.
 Settings for the odd lines:

  BPL1MOD =  0   ->
               $mmmm
                =  0
  BPL2MOD = -8   ->
               $nnnn
                = -8

 We're now on an even line;

 Playfield 1, after a whole line has been fetched, is in this situation:
 the BPLxPTs point to the next line plus ofs+8+8, which, being the next
 line odd, is 16 bytes beyond the offset desired, so the modulo must
 be -16.
 Therefore, the modulo settings for the even lines are:

  BPL1MOD = -16   ->
               $pppp
                = -16
  BPL2MOD = -8    ->
               $qqqq
                = -8
```

1.38 3.5.3.2.3 XPos (64) Belongs to {0,61,62,63}

```
                3.5.3.2.3   XPos (64) Belongs to {0,61,62,63}




               In this case
                the playfield 1 has the same offset independently from the
line: by setting it to
               offset(XPos)
               +8, it's enough to have a modulo of -8
to "recover" the "extra-fetched" bytes on both the even and odd lines.

To find the right values for  BPL2MOD, we follow the process of DMA data
fetching step-by-step:

 starting from odd line;
 base offset: ofs =
               offset(XPos)
                 playfield 2 offset: ofs2 =
               offset(XPos+4)
                =
```

```
               offset(XPos)
                = ofs;

 After fetching a whole line of playfield 2, the BPLxPTs point to the
 start of the next line plus the offset plus 8 of "extra-fetched" bytes,
 i.e. the start of the next line plus ofs2+8 that is equal to ofs+8;
 since the next line is even and since playfield 2 has ofs+8 as offset on
 those lines, no more bytes have to be skipped and the modulo is 0.
 Therefore, the modulo settings for the odd lines are:

  BPL1MOD = -8   ->
               $mmmm
                = -8
  BPL2MOD =  0   ->
               $nnnn
                =  0

 We're now on an even line;

 Playfield 2, after a whole line has been fetched, is in this situation:
 its BPLxPTs point at the start of the next line plus ofs+8+8, which,
 given that the next line is odd, is 16 bytes beyond the offset desired,
 so a modulo of -16 is simply what is needed to "recover" those bytes.
 Therefore, the modulo settings for the even lines are:

  BPL1MOD = -8    ->
               $pppp
                = -8
  BPL2MOD = -16   ->
               $qqqq
                = -16
```

1.39 3.5.3.2.4 Settings Summary

```
                3.5.3.2.4   Settings Summary


Summarizing the results found so far:

 we have a copperlist made of DsplHt/2 (DsplHt = display height in lines)

               chunks of this kind
               , preceded, possibly among other instructions, by the
 COPMOVEs to the BPLxPT registers.
 Depending on the horizontal position desired (XPos), the fields of the
 copperlist have to be filled as follows:

 BPLxPT:

  playfield   BPLxPT

  2
               GetBPLxPT(XPos)
                  1
```

```
                GetBPLxPT(XPos+4)
                  BPLCON1:

lines   BPLCON1

odd
                $ssss
                 =
                GetBPLCON1(XPos)
                   even
                $tttt
                 =
                GetBPLCON1(XPos+4)
                  BPLxMOD:

lines   playfield   XPos (64) in    BPLxMOD

odd     1
                {1,...,56}

                $mmmm
                 = -8
odd     2
                {1,...,56}

                $nnnn
                 = -8
even    1
                {1,...,56}

                $pppp
                 = -8
even    2
                {1,...,56}

                $qqqq
                 = -8

odd     1
                {57,...,60}

                $mmmm
                 =  0
odd     2
                {57,...,60}

                $nnnn
                 = -8
even    1
                {57,...,60}

                $pppp
                 = -16
even    2
                {57,...,60}

                $qqqq
```

```
                = -8

odd     1
             {0,61,62,63}

             $mmmm
              = -8
odd     2
             {0,61,62,63}

             $nnnn
              =  0
even    1
             {0,61,62,63}

             $pppp
              = -8
even    2
             {0,61,62,63}

             $qqqq
              = -16
```

- if we had strictly followed the directions given until this point, the BPLxPTs should have been:

```
 start line   playfield   BPLxPT

 odd           2
            GetBPLxPT(XPos)
                 odd          1
            GetBPLxPT(XPos+4)
                 even         2
            GetBPLxPT(XPos+4)
                 even         1
            GetBPLxPT(XPos+8)
                but this is *not* needed because the odd/even definitions  ↩
                   given here are
just mere conventions for the sake of readability
```

1.40 3.6 Cross Playfield Mode

```
                3.6   Cross Playfield Mode
```

Whatever kind of display we have examined up to this point always three or even four bitplanes were unused. What a waste. If you think this, then I agree with you. Theorically we could use those planes, for example, to open 16-bit displays, but unluckily a HalfRes-like mode would not be possible (the ChipSet offers "only" two indipendent horizontal scroll values) and, even if it had been, each pixel would have looked like four LORES

pixels (and 3 of them would have been "averages"). FullRes instead, would be possible, but actually the cost of the conversion would be so high to make it almost unfeasible. Yet, there's still something we can do: why don't we use those planes for another playfield? Yes, this can be actually obtained without much effort from the CPU, some more DMA work and some differences respect to a real Dual Playfield (like the Amiga's).

1.41 3.6.1 Limitations

3.6.1 Limitations

Before going further we have a look at the limitations: the two playfields share the settings of BPLxMOD, DDFSTxx and BPLCON1, so they must have the same:

- horizontal resolution (both FullRes or both HalfRes)
- display width
- [HalfRes] screen width
- [HalfRes] screen horizontal position
- [HalfRes] horizontal scroll, chequer settings

other limitations are:

- loss of many (175) colors for pixel-value-based transparency of front playfield
- [HalfRes] MskPln is compulsory in Dual mode ("uncotrolled" average pixels may yield one of those "lost" colors

Actually, in FullRes most limitations could be overcome by using Amiga's own Dual Playfield mode; though this would require a major rework of the definitions and of all routines written to handle all the other modes.

1.42 3.6.2 BitPlanes Assignment

```
3.6.2   BitPlanes Assignment


After allocating two more CHIP ram buffers for the front playfield planes,
we assign them the BPLxPT registers as follows:

 (FPfldPlnX = Front Playfield Plane X)

 - Fullres or HalfRes without MskPln:

   BPL1PT = VdoPln0 address
   BPL2PT = VdoPln0 [HalfRes] or VdoPln1 [FullRes] address
   BPL3PT = SlcPln0 address
   BPL4PT = SlcPln1 address
   BPL5PT = FPfldPln0 address
   BPL6PT = FPfldPln1 address

   (first 4 assignments unchanged)

 - HalfRes with MskPln

   BPL1PT = VdoPln0 address
   BPL2PT = VdoPln0 address
   BPL3PT = SlcPln0 address
   BPL4PT = SlcPln1 address
   BPL5PT = MskPln address
   BPL6PT = FPfldPln0 address
   BPL7PT = FPfldPln1 address

   (first 5 assignments unchanged)

 in this way we use the SlcPlns and the MskPln also for the front play-
 field, thus we can avoid to allocate other DMA/memory -consuming planes.
```

1.43 3.6.3 Palette Settings

```
                3.6.3   Palette Settings


In the
               first section
                we had to acknowledge the limitations of our mode;
now it's time to see its incredible advantages.
After
               assigning the bitplanes
               , we now see how the palette settings change
according to those assignments, examining for a start the simplest case
(HalfRes without MskPln or FullRes). Before beginning, it's important to
point out that there is no reason to force the two playfields to have same
RGBx mode - yes, each playfield can have its own palette of 256 colors!

There are several ways to look at this problem and personally I found that
```

the less puzzling is considering the components separately:

```
 component:   CN

 FPfldPln1:   F
 FPfldPln0:   f
 SlcPln1  :   S
 SlcPln0  :   s
 VdoPln1  :   V
 VdoPln0  :   v
```

our task here is to find the 24-bit RGB values to write to the 64 color registers indexed by %FfSsVv.
But first we have to ask ourselves: what does %FfSsVv represent?
It indicates the color register that is selected when the component CN (selected by %Ss) of a pixel on the front playfield with CV=%Ff is superimposed to the component CN of a pixel with CV=%Vv on the back playfield.
Secondly, the question is: what happens to that component? Is it completely hidden by the front playfield's?
Well, surely not; in that case, in fact, we would have that the back playfield CN component of every pixel is always hidden and, by extending this reasoning to the other compoents, the back playfield is always hidden by the back one, whatever value their pixels have - pretty useless.
Instead, we should "merge" the two components together, specifying their "weight" in the final outcome. We define this "weight" as the "opacity" of the front playfield, which is a measure of how much transparent it is. Here we'll call it "o" and make it range from 0 (totally transparent) to 256 totally opaque.
So the resulting component intensity becomes: C=(C0*(256-o))+(C1*o), where C0=intensity of component CN from back playfield and C1=intensity of component CN from front playfield (these two values are totally indipendent, and that's why the two playfields need not to have the same RGBx mode).
All that remains to be done is just writing C to the COLORxx register, where xx is selected by %fSsVv and the bank (bits 15-13 of BPLCON3) it belongs to is selected by %F.

Now let's also consider the case of HalfRes with MskPln:

```
 component:   CN

 FPfldPln1:   F
 FPfldPln0:   f
 MskPln   :   M
 SlcPln1  :   S
 SlcPln0  :   s
 VdoPln1  :   V
 VdoPln0  :   v
```

things, fortunately, don't change much: of course the number of registers to set is doubled but, apart from the different selection of COLORxx registers (xx=%MSsVv; bank=%Ff) the principle and the calculations are exactly the same.

- it’s evident that in this way it’s very easy to produce cross-fading effects (that’s where the name of the mode came from...)
- a drawback of this method is that we can’t pick an RGBx value as "always transparent" (like color 0 in Amiga’s Dual Playfield); a partial solution is given in the following section

1.44 3.6.4 Dual Modality

3.6.4 Dual Modality

The Cross Playfield looks nice, though it lacks of a color which the back playfield can be seen through without color alteration, like in a normal Dual Playfield happens. Yet, with a bit more or patience, we can "emulate" this feature, too.

Let’s say that we want the color %RrBbGgXx to be completely transparent and that the data on the bitplanes is arranged in this way:

```
 FPfldPln1:   RGBX
 FPfldPln0:   rgbx
 SlcPln1  :   SSSS
 SlcPln0  :   ssss
 VdoPln1  :   UVWY
 VdoPln0  :   uvwy
```

considering only RN, what we want here is that the RGB color associated to %RrSsUu is exactly the same of %SsUu in the RGBx mode used by the back playfield: in other words, the RGB value to write to COLORxx (xx=%RrSsUu) is the same we use for %SsUu for back playfield in a non-Cross Playfield mode. Doing the same for the other components is all that remains to be done to reach our goal.

Unfortunately this method has a considerable "dark side": what happens to color %Rr0000 (different from %RrGgBbXx), for example? The components GN, BN and xN are OK, but RN is treated as transparent, so the final outcome is that the color is uncorrectly shown on the screen. Of course, this holds true for all the possible colors whose RV is %Rr and applies also to all the other components, so, in the end, we have that all the colors who have at least one CV equal to one in %RrGgBbXx look bad. This means that the number of unaffected colors is reduced to 3^4 = 81 (we can freely select three CVs - instead of four - per component).

Despite the loss of colors, we have now a real Dual Playfield mode, with the additional feature that the non-transparent colors have variable opacity! Not even Amiga’s Dual Playfield mode can do this!

1.45 3.7 Screen Buffering

3.7 Screen Buffering

Nothing of what we have seen so far (and we'll see later) stops us from creating buffered TCS displays... so if your game or your demo (or whatever) needs to keep itself synchronized with the video refresh and wants to boast flicker/jerks -free graphics, we just have to think a bit about how we can double/triple buffer.

Explaining here any buffering method would be superfluous, whereas it is important to point out all the TCS-related topics.

First of all, _what_ exactly do we have to buffer?
In HalfRes mode, we have to reserve two or three buffers for the only VdoPln available (VdoPln0), which, as you should remember, is used as the ChnkScr; in FullRes mode, both VdoPln0 and VdoPln1 must be buffered, while ChnkScr, which is a separate buffer (preferably in FAST memory), must not: in fact, after the
ChnkScr -> VdoPlns conversion
, the ChnkScr can be used
again as the buffering is applied to the buffers that are actually shown on the monitor.

Another important thing to consider is that the
scroll
settings must
affect only the current physical screen, so also the copperlists must be buffered (it's not enough to change the pointers to the planes in a single copperlist).